Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.В. ГУНЬКО

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Конспект лекций

НОВОСИБИРСК
2011

УДК 004.45(075.8)
Г 948

Рецензенты: *А.А. Малявко*, канд. техн. наук, доцент,
А.Б. Колкер, канд. техн. наук, доцент

Работа подготовлена на кафедре автоматики
Новосибирского государственного технического университета
для студентов 4 курса
по направлению 220200 «Автоматизация и управление»

Гунько А.В.

Г 948 Системное программное обеспечение : Конспект лекций / А.В. Гунько. – Новосибирск : Изд-во НГТУ, 2011.– 138 с.

ISBN 978-5-7782-1670-9

В конспекте лекций изложены основные сведения об организации операционных систем и сред, обсуждаются методы и средства разработки многозадачного и многопоточного программного обеспечения в операционных системах семейства Windows и Linux, а также средства межзадачной и межпоточной коммуникации: неименованные и именованные каналы, семафоры, очереди сообщений, разделяемая память, взаимные исключения и условные переменные.

Конспект лекций может быть полезен студентам и аспирантам ряда других технических специальностей, связанных с разработкой многозадачного и многопоточного программного обеспечения.

УДК 004.45(075.8)

ISBN 978-5-7782-1670-9

1. ОПЕРАЦИОННЫЕ СИСТЕМЫ И СРЕДЫ

1.1. ОПРЕДЕЛЕНИЕ И СОСТАВ СИСТЕМНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Под системным программным обеспечением (*System Software*) понимаются программы и комплексы программ, являющиеся общими для всех, кто совместно использует технические средства компьютера, и применяемые как для автоматизации разработки новых программ, так и для организации выполнения программ существующих. С этих позиций системное программное обеспечение может быть разделено на следующие пять групп:

- Операционные системы (ОС);
- Системы управления файлами;
- Интерфейсные оболочки для взаимодействия пользователя с ОС и программные среды;
- Системы программирования;
- Утилиты.

Рассмотрим вкратце эти группы системных программ.

1.1.1. Операционные системы

Под *операционной системой* понимают комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой – предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надежных вычислений [1]. Любой из компонентов прикладного программного обеспечения обязательно работает под управлением ОС. На рис. 1.1 изображена обобщенная структура программного обеспечения вычислительной системы. Видно, что ни один из компонентов программного обеспечения, за исключением самой ОС, не имеет непосредственного доступа к аппаратуре компьютера. Даже пользователи взаимодействуют со своими программами через интерфейс ОС. Любые их команды, прежде чем попасть в прикладную программу, сначала проходят через ОС.

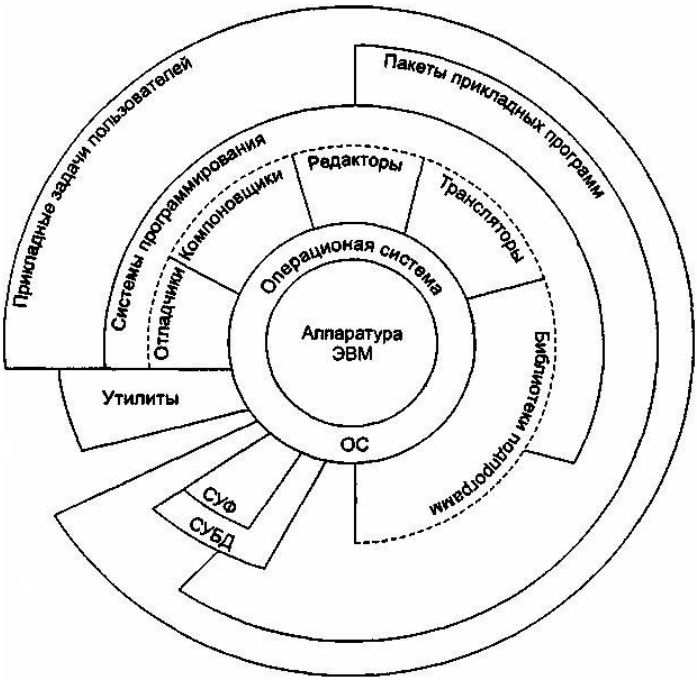

Рис. 1.1. Структура системного программного обеспечения

Основными функциями, которые выполняет ОС, являются следующие:

- прием от пользователя (или от оператора системы) заданий или команд, сформулированных на соответствующем языке – в виде директив (команд) оператора или в виде указаний (своеобразных команд) с помощью соответствующего манипулятора (например, с помощью мыши), – и их обработка;
- прием и исполнение программных запросов на запуск, приостановку, остановку других программ;
- обеспечение работы систем управлений файлами (СУФ) и/или систем управления базами данных (СУБД), что позволяет резко увеличить эффективность всего программного обеспечения;
- обеспечение режима мультипрограммирования, то есть выполнение двух или более программ на одном процессоре, создающее видимость их одновременного исполнения;

- обеспечение функций по организации и управлению всеми операциями ввода/вывода;
- удовлетворение жестким ограничениям на время ответа в режиме реального времени (характерно для соответствующих ОС);
- распределение памяти, а в большинстве современных систем и организация виртуальной памяти;
- планирование и диспетчеризация задач в соответствии с заданными стратегией и дисциплинами обслуживания;
- организация механизмов обмена сообщениями и данными между выполняющимися программами;
- защита одной программы от влияния другой; обеспечение сохранности данных;
- предоставление услуг на случай частичного сбоя системы;
- обеспечение работы систем программирования, с помощью которых пользователи готовят свои программы.

1.1.2. Системы управления файлами

Назначение *системы управления файлами* – организация более удобного доступа к данным, организованным как файлы. Именно благодаря системе управления файлами вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нем.

Выделение этого вида системного программного обеспечения в отдельную категорию представляется целесообразным, поскольку ряд ОС позволяет работать с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно). В этом случае говорят о монтируемых файловых системах, и в этом смысле они самостоятельны. Более того, можно назвать примеры простейших ОС, которые могут работать и без файловых систем, а значит, им не обязательно иметь систему управления файлами, либо они могут работать с одной из выбранных файловых систем.

1.1.3. Интерфейсные оболочки

Основное назначение *интерфейсных оболочек* – прием от пользователя команд, сформулированных на соответствующем языке в виде команд оператора, или в виде указаний (своеобразных команд)

с помощью графического манипулятора. Интерфейсные оболочки могут также расширять возможности по управлению ОС, либо изменять встроенные в систему возможности. Интерфейсные оболочки могут быть текстовыми (NC, VC для DOS, FAR для Windows, MC для UNIX/Linux) и графическими (KDE, Gnome для Linux).

Интерфейс также необходим программам для обращения к ОС с целью получить определенный сервис – выполнить операцию ввода/вывода, получить или освободить участок памяти и т. д. В случае, если программа реализована для другой ОС, то для ее нормальной работы в текущей ОС нужны соответствующие средства:

- VDM (Virtual DOS machine) для выполнения DOS-программ в Windows 9x/NT/2000;
- WINE для выполнения Windows-программ в Linux.

Существуют также программы-эмуляторы, позволяющие смоделировать в одной операционной системе какую-либо другую операционную систему (VMWare Workstation, VirtualPC, VirtualBox).

1.1.4. Системы программирования

Система программирования представлена такими компонентами, как транслятор с соответствующего языка программирования, библиотеки подпрограмм, редакторы, компоновщики и отладчики.

Не бывает самостоятельных (оторванных от ОС) систем программирования. Любая система программирования может работать только в соответствующей ОС, под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС. Так, коммерческая система программирования Kylix на языке C/C++ от фирмы Borland (ныне не поддерживаемая) и инструментарий разработки ПО на языке программирования C++ Qt 4.x, распространяемый по лицензии GNU GPL, позволяют получать программы, разрабатываемые в Windows, в версиях и для Windows и для Linux.

В том случае, когда создаваемые программы должны работать совсем на другой аппаратной базе, говорят о кросс-системах. Так, для ПК на базе микропроцессоров семейства i80x86 имеется большое количество кросс-систем, позволяющих создавать программное обеспечение для различных микропроцессоров и микроконтроллеров.

1.1.5. Утилиты

Утилиты – специальные системные программы, с помощью которых можно как обслуживать саму операционную систему, так и подготавливать для работы носители данных, выполнять перекодирование данных, осуществлять оптимизацию размещения данных на носителе и производить некоторые другие работы, связанные с обслуживанием вычислительной системы. К утилитам следует отнести и программу разбиения накопителя на магнитных дисках на разделы, и программу форматирования.

Естественно, что утилиты могут работать только в соответствующей операционной среде.

1.2. ОПЕРАЦИОННАЯ СРЕДА

Операционная система выполняет функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы вычислительной системы между различными процессами и образует программную среду, в которой выполняются прикладные программы пользователей. Такая среда называется операционной.

Любая программа имеет дело с некоторыми исходными данными, которые она обрабатывает, и порождает в конечном итоге некоторые выходные данные, результаты вычислений. Развитие системного программирования и самого системного программного обеспечения пошло по пути выделения наиболее часто встречающихся операций и создания для них соответствующих программных модулей, которые можно в дальнейшем использовать в большинстве вновь создаваемых программ. Таким образом, можно сказать, что термин «операционная среда» означает, прежде всего, соответствующие интерфейсы, необходимые программам и пользователям для обращения к ОС с целью получить определенные сервисы.

Параллельное существование терминов «операционная система» и «операционная среда» вызвано тем, что ОС в общем случае может поддерживать несколько операционных сред.

Подводя итог, можно сказать, что операционная среда – это то системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды.

1.3. ПОНЯТИЯ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА И РЕСУРСА

Понятие *вычислительный процесс* (или просто – *процесс)* является одним из основных при рассмотрении операционных систем. Последовательный процесс (иногда называемый *задачей)* – это выполнение отдельной программы с ее данными на последовательном процессоре.

Изначально *задача (task)* определялась, как совокупность связанных между собой и образующих единое целое программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации. Затем задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач.

Определение концепции процесса преследует цель выработать механизмы распределения и управления ресурсами. Термин *ресурс* определяет повторно используемые, относительно стабильные и часто недостающие потребителям объекты, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, ресурсом называется всякий объект, который может распределяться внутри системы.

Ресурсы бывают разделяемыми, когда несколько процессов могут их использовать одновременно (в один и тот же момент времени) или параллельно (попеременно), а могут быть и неделимыми (рис. 1.2).

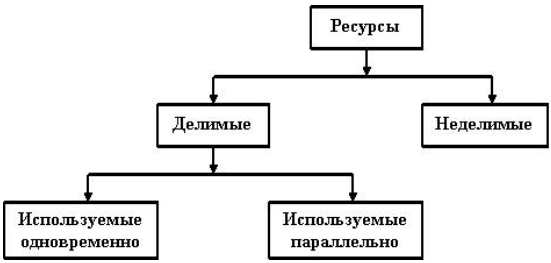

Рис. 1.2. Классификация ресурсов

В первых вычислительных системах программа могла выполняться только после полного завершения предыдущей. Центральный процессор осуществлял и выполнение вычислений, и управление операциями ввода/вывода данных. Соответственно, пока осуществлялся обмен данными

между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления.

Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и вычисления на центральном процессоре. Однако все равно процессор продолжал часто и долго простаивать, дожидаясь завершения очередной операции ввода/вывода.

Было предложено организовать *мультипрограммный* (мультизадачный) режим работы вычислительной системы. Суть его в том, что пока одна задача ожидает завершения очередной операции ввода/вывода, другая задача может быть поставлена на решение.

Из рис. 1.3 видно, что благодаря совмещению во времени выполнения двух программ общее время выполнения двух задач получается меньше, чем если бы мы выполняли их по очереди (запуск одной только после полного завершения другой), Из этого же рисунка видно, что время выполнения каждой задачи в общем случае становится больше, чем если бы мы выполняли каждую из них как единственную.

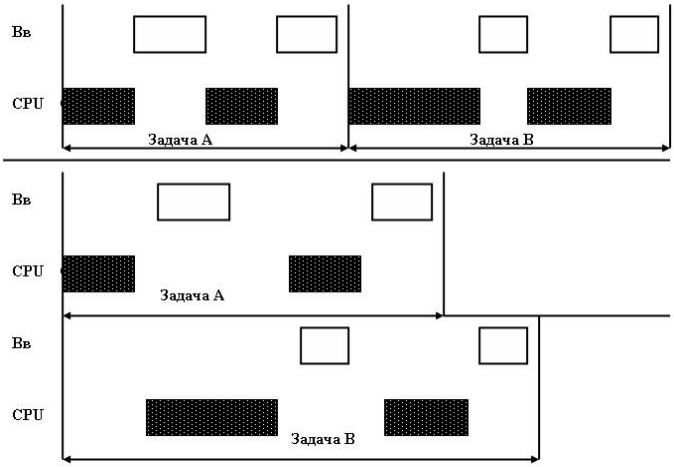

Рис. 1.3. Пример выполнения двух программ
в однопрограммном и мультипрограммном режимах

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме

(всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

При необходимости использовать какой-либо ресурс (оперативную память, устройство ввода/вывода, массив данных и т. п.) задача обращается к супервизору операционной системы – ее центральному управляющему модулю, который может состоять из нескольких модулей, например: супервизор ввода/вывода, супервизор прерываний, диспетчер задач и т. д. – посредством специальных вызовов (команд, директив) и сообщает о своем требовании.

При этом указывается вид ресурса и, если надо, его объем (например, количество адресуемых ячеек оперативной памяти, количество дорожек или секторов на системном диске, устройство печати и объем выводимых данных и т. п.).

Директива обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы, если такой существует. Большинство вычислительных комплексов имеют два режима работы: привилегированный (режим супервизора) и пользовательский.

Ресурс может быть выделен задаче, обратившейся к супервизору с соответствующим запросом, если:

- он свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;
- текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;
- ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, операционная система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя).

После окончания работы с ресурсом задача с помощью специального вызова супервизора сообщает операционной системе об отказе от ресурса, или операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции.

Супервизор операционной системы, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть – в зависимости от приня-

той *дисциплины обслуживания* (правила обслуживания) и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению. После этого управление либо передается данной задаче, либо возвращается той, которая только что освободила ресурс.

1.4. ДИАГРАММА СОСТОЯНИЙ ПРОЦЕССА

Необходимо различать системные управляющие процессы, представляющие работу супервизора операционной системы и занимающиеся распределением и управлением ресурсов, от других процессов:

- системных обрабатывающих процессов, которые не входят в ядро операционной системы;
- процессов пользователя.

Процесс может находиться в активном и пассивном состоянии. В *активном состоянии* процесс может участвовать в конкуренции за использование ресурсов вычислительной системы, а в пассивном – он только известен системе, но в конкуренции не участвует (хотя его существование в системе и сопряжено с предоставлением ему оперативной и/или внешней памяти).

Активный процесс может быть в одном из следующих состояний:

- *выполнения* – все затребованные процессом ресурсы выделены. В этом состоянии в каждый момент времени может находиться только один процесс, если речь идет об однопроцессорной вычислительной системе;
- *готовности к выполнению* – ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения;
- *блокирования* или *ожидания* – затребованные ресурсы не могут быть предоставлены, или не завершена операция ввода/вывода.

В большинстве операционных систем последнее состояние, в свою очередь, подразделяется на множество состояний ожидания, соответствующих определенному виду ресурса, из-за отсутствия которого процесс переходит в заблокированное состояние.

За время своего существования процесс может неоднократно совершать переходы из одного состояния в другое. Возможные переходы процесса из одного состояния в другое отображены в виде графа состояний на рис. 1.4.

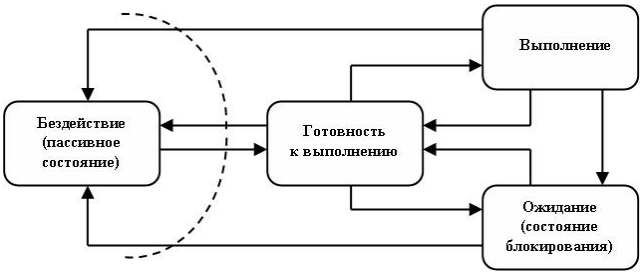

Рис. 1.4. Граф состояний процесса

Процесс из состояния бездействия переходит в состояние готовности в следующих случаях:

- по команде оператора (пользователя). Имеет место в тех диалоговых операционных системах, где программа может иметь статус задачи (и при этом являться пассивной), а не просто быть исполняемым файлом и только на время исполнения получать статус задачи;
- при выборе из очереди планировщиком (характерно для операционных систем, работающих в пакетном режиме);
- по вызову из другой задачи (посредством обращения к супервизору один процесс может создать, инициировать, приостановить, остановить, уничтожить другой процесс);
- по прерыванию от внешнего инициативного устройства (сигнал о свершении некоторого события может запускать соответствующую задачу). Устройство называется *«инициативным»,* если по сигналу запроса на прерывание от него должна запускаться некоторая задача;
- при наступлении запланированного времени запуска программы.

Из состояния выполнения процесс может выйти по одной из следующих причин:

- процесс завершается, при этом он посредством обращения к супервизору передает управление операционной системе и сообщает о своем завершении. В результате этих действий супервизор либо переводит его в список бездействующих процессов (процесс переходит в пассивное состояние), либо уничтожает (уничтожается не сама программа, а именно задача, которая соответствовала исполнению некоторой программы). В состояние бездействия процесс может быть переведен принудительно: по команде оператора (действие этой и других команд оператора реализуется системным процессом, который «транслирует» команду в запрос к супервизо-

ру с требованием перевести указанный процесс в состояние бездействия), или путем обращения к супервизору операционной системы из другой задачи с требованием остановить данный процесс;

- процесс переводится супервизором операционной системы в состояние готовности к исполнению в связи с появлением более приоритетной задачи или в связи с окончанием выделенного ему кванта времени;
- процесс блокируется (переводится в состояние ожидания) либо вследствие запроса операции ввода/вывода (которая должна быть выполнена прежде, чем он сможет продолжить исполнение), либо в силу невозможности предоставить ему ресурс, запрошенный в настоящий момент (причиной перевода в состояние ожидания может быть и отсутствие сегмента или страницы в случае организации механизмов виртуальной памяти, а также по команде оператора на приостановку задачи или по требованию через супервизор от другой задачи).

При наступлении соответствующего события (завершилась операция ввода/вывода, освободился затребованный ресурс, в оперативную память загружена необходимая страница виртуальной памяти и т. д.) процесс деблокируется и переводится в состояние готовности к исполнению.

1.5. РЕАЛИЗАЦИЯ ПОНЯТИЯ ПОСЛЕДОВАТЕЛЬНОГО ПРОЦЕССА В ОС

Для того чтобы операционная система могла управлять процессами, она должна располагать всей необходимой для этого информацией. С этой целью на каждый процесс заводится специальная информационная структура, называемая *дескриптором процесса* (описателем задачи, блоком управления задачей). В общем случае дескриптор процесса содержит следующую информацию:

- идентификатор процесса (так называемый PID – process identificator);
- тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;
- приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы. В рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы;

- переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, в состоянии выполнения, ожидание устройства ввода/вывода и т. д.);
- защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процессора, если процесс прерывается, не закончив работы. Эта информация называется *контекстом задачи;*
- информацию о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода/вывода и т. п.);
- место (или его адрес) для организации общения с другими процессами;
- параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);
- в случае отсутствия системы управления файлами – адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если ее вытесняет другая.

Описатели задач, как правило, постоянно располагаются в оперативной памяти с целью ускорить работу супервизора, который организует их в списки (очереди) и отображает изменение состояния процесса перемещением соответствующего описателя из одного списка в другой. Для каждого состояния (за исключением состояния выполнения для однопроцессорной системы) операционная система ведет соответствующий список задач, находящихся в этом состоянии. Однако для состояния ожидания может быть не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания.

1.6. ПРОЦЕССЫ И ПОТОКИ

Понятие процесса было введено для реализации идей мультипрограммирования. Для реализации мультизадачности необходимо было тоже ввести соответствующую сущность. Такой сущностью и стали так называемые «легковесные» процессы, или, как их преимущественно называют, – *потоки* или *треды* (*thread* – поток, нить).

Когда говорят о *процессах* (*process*), то тем самым хотят отметить, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы – файлы, окна, семафоры и т. д. Такая обо-

собленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы вычислительной системы, конкурируют друг с другом. В общем случае процессы просто никак не связаны между собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную систему.

Однако желательно иметь еще и возможность задействовать внутренний параллелизм, который может быть в самих процессах. Такой внутренний параллелизм встречается достаточно часто и его использование позволяет ускорить их решение. Например, некоторые операции, выполняемые приложением, могут требовать для своего исполнения достаточно длительного использования центрального процессора. В этом случае при интерактивной работе с приложением пользователь вынужден долго ожидать завершения заказанной операции и не может управлять приложением до тех пор, пока операция не выполнится до самого конца. Такие ситуации встречаются достаточно часто, например, при обработке больших изображений в графических редакторах. Если же программные модули, исполняющие такие длительные операции, оформлять в виде самостоятельных «подпроцессов» (легковесных или облегченных процессов – потоков), которые будут выполняться параллельно с другими «подпроцессами» (потоками), то у пользователя появляется возможность параллельно выполнять несколько операций в рамках одного приложения (процесса). Легковесными эти задачи называют потому, что операционная система не должна для них организовывать полноценную виртуальную машину. Эти задачи не имеют своих собственных ресурсов, они развиваются в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, что и данный процесс. Единственное, что им необходимо иметь, – это процессорный ресурс. В однопроцессорной системе потоки разделяют между собой процессорное время так же, как это делают обычные процессы, а в мультипроцессорной системе могут выполняться одновременно, если не встречают конкуренции из-за обращения к иным ресурсам.

Главное, что обеспечивает многопоточность, – это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Программа, оформленная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет параллельного выполнения отдельных ее частей.

Сущность «процесс» предполагает, что при диспетчеризации нужно учитывать все ресурсы, закрепленные за ним. А при манипулировании потоками можно менять только контекст задачи, если мы переключаемся с одной задачи на другую в рамках одного процесса. Все остальные вычислительные ресурсы при этом не затрагиваются. Каждый процесс всегда состоит, по крайней мере, из одного потока, и только если имеется внутренний параллелизм, программист может «расщепить» один процесс на несколько параллельных потоков.

Каждый поток выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Потоки, как и процессы, могут порождать потоки-потомки, поскольку любой процесс состоит по крайней мере из одного потока. Подобно традиционным процессам (то есть процессам, состоящим из одного потока) каждый поток может находиться в одном из активных состояний. Пока один поток заблокирован (или просто находится в очереди готовых к исполнению задач), другой поток того же процесса может выполняться. Потоки разделяют процессорное время так же, как это делают обычные процессы, в соответствии с различными вариантами диспетчеризации.

Все потоки имеют одно и то же виртуальное адресное пространство своего процесса. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к каждому виртуальному адресу, один поток может использовать стек другого потока. Между потоками нет полной защиты, так как это, во-первых, невозможно, а во-вторых, не нужно. Кроме того, все потоки разделяют также набор открытых файлов, используют общие устройства, выделенные процессу, имеют одни и те же наборы сигналов, семафоры и т. п. А что у потоков будет их собственным? Собственными являются программный счетчик, стек, рабочие регистры процессора, локальные переменные, потоки-потомки, состояние.

Вследствие того, что потоки, относящиеся к одному процессу, выполняются в одном и том же виртуальном адресном пространстве, между ними легко организовать тесное взаимодействие, в отличие от процессов, для которых нужны специальные механизмы обмена сообщениями и данными. Более того, программист, создающий многопоточное приложение, может заранее продумать работу множества потоков процесса таким образом, чтобы они могли взаимодействовать наиболее выгодным способом, а не участвовать в конкуренции за предоставление ресурсов тогда, когда этого можно избежать.

1.7. УПРАВЛЕНИЕ ЗАДАЧАМИ В ОС

Время центрального процессора и оперативная память являются основными ресурсами в случае реализации многозадачных вычислений. Способы распределения времени центрального процессора сильно влияют на скорость выполнения отдельных вычислений, и на общую эффективность вычислительной системы. От выбранных механизмов распределения памяти между выполняющимися процессорами тоже очень сильно зависит и эффективность использования ресурсов системы, и ее производительность.

В данном вопросе не будем разделять понятия процесс (process) и поток (thread), вместо этого используя как бы обобщающий термин *task* (задача). Операционная система выполняет следующие основные функции, связанные с управлением задачами:

- создание и удаление задач;
- планирование и диспетчеризация задач;
- синхронизация задач, обеспечение их средствами коммуникации.

Система управления задачами обеспечивает прохождение их через компьютер. В зависимости от состояния процесса ему должен быть предоставлен тот или иной ресурс. Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Созданный новый процесс необходимо разместить в основной памяти – следовательно, ему необходимо выделить часть адресного пространства. Новый порожденный поток текущего процесса необходимо включить в общий список задач, конкурирующих между собой за ресурсы центрального процессора.

Задача может породить новую задачу. При этом между процессами появляются «родственные» отношения. Порождающая задача называется «предком» или «родителем», а порожденная – «потомком» или «дочерней задачей». «Предок» может приостановить или удалить свою дочернюю задачу, тогда как «потомок» не может управлять «предком».

Очевидно, что на распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно. Например, всем выполняющимся процессам требуется некоторое устройство с последовательным доступом. Но поскольку оно не может

распределяться между параллельно выполняющимися процессами, то процессы вынуждены будут очень долго ждать своей очереди.

Если подобрать набор таких процессов, которые не будут конкурировать между собой за неразделяемые ресурсы при параллельном выполнении, то, скорее всего, процессы смогут выполниться быстрее (из-за отсутствия дополнительных ожиданий), да и имеющиеся в системе ресурсы будут использоваться более эффективно.

Задача подбора такого множества процессов, что при выполнении они будут как можно реже конфликтовать из-за имеющихся в системе ресурсов, называется *планированием вычислительных процессов.*

Задача планирования процессов возникла очень давно – в первых пакетных ОС. В настоящее время актуальность этой задачи не так велика. На первый план вышли задачи динамического (или краткосрочного) планирования, то есть текущего наиболее эффективного распределения ресурсов, возникающего практически при каждом событии. Задачи динамического планирования стали называть *диспетчеризацией.*

При рассмотрении стратегий планирования, как правило, идет речь о краткосрочном планировании, то есть о диспетчеризации. *Стратегия планирования* определяет, какие процессы мы планируем на выполнение для того, чтобы достичь поставленной цели. Известно большое количество различных стратегий выбора процесса, которому необходимо предоставить процессор. Среди них прежде всего можно назвать следующие стратегии:

- по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;
- отдавать предпочтение более коротким процессам;
- предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Когда говорят о стратегии обслуживания, всегда имеют в виду понятие процесса, а не понятие задачи, поскольку процесс, как мы уже знаем, может состоять из нескольких потоков (задач).

Когда говорят о диспетчеризации, то всегда в явном или неявном виде имеют в виду понятие задачи (потока).

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач, различают два больших класса дисциплин обслуживания – *бесприоритетные и приоритетные.*

При *бесприоритетном* обслуживании выбор задач производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания.

При реализации *приоритетных* дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения.

Перечень дисциплин обслуживания и их классификация приведены на рис. 1.5.

Рассмотрим кратко некоторые основные (наиболее часто используемые) дисциплины диспетчеризации.

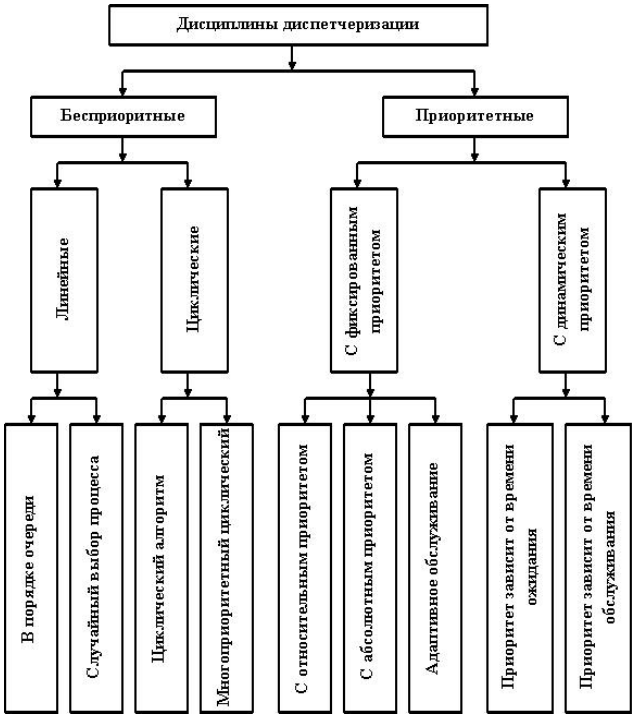

Рис. 1.5. Дисциплины диспетчеризации

Самой простой в реализации является **дисциплина FCFS** (first come – first served), согласно которой задачи обслуживаются «в порядке очереди», то есть в порядке их появления. Те задачи, которые были

заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь готовности перед теми задачами, которые еще не выполнялись.

Другими словами, образуются две очереди (см. рис. 1.6): одна очередь образуется из новых задач, а вторая очередь – из ранее выполнявшихся, но попавших в состояние ожидание.

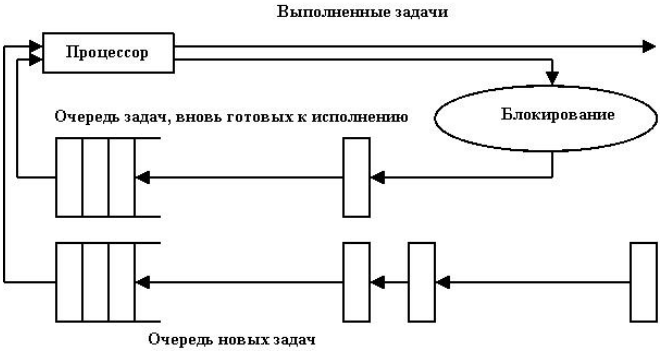

Рис. 1.6. Дисциплина диспетчеризации FCFS

К достоинствам этой дисциплины прежде всего можно отнести простоту реализации и малые расходы системных ресурсов на формирование очереди задач. Однако эта дисциплина приводит к тому, что при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать столько же, сколько и трудоемкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT.

Дисциплина обслуживания SJN (shortest job next: следующим будет выполняться кратчайшее задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость сообщать ОС характеристики задач, в которых описывались бы потребности в ресурсах вычислительной системы, привела к тому, что были разработаны соответствующие языковые средства.

Пользователи должны были указывать предполагаемое время выполнения, и чтобы они не пытались указать заведомо меньшее время выполнения, ввели подсчет реальных потребностей. Диспетчер задач

сравнивал заказанное время и время выполнения, и в случае превышения указанной оценки в данном ресурсе ставил данное задание не в начало, а в конец очереди.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. И задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводит к тому, что задания, которым требуется очень немного времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами, что не всегда хорошо.

Для устранения этого недостатка и была предложена **дисциплина SRT** (shortest remaining time: следующее задание требует меньше всего времени для своего завершения).

Все эти три дисциплины обслуживания могут использоваться для пакетных режимов обработки. Для интерактивных же вычислений желательно прежде всего обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мультитерминальной. Если же это однопользовательская система, но с возможностью мультипрограммной обработки, то желательно, чтобы те программы, с которыми мы сейчас непосредственно работаем, имели лучшее время реакции, нежели наши фоновые задания. Для решения подобных проблем используется дисциплина обслуживания, называемая RR (round robin, круговая, карусельная), и приоритетные методы обслуживания.

Дисциплина обслуживания RR предполагает, что каждая задача получает процессорное время порциями (говорят: квантами времени, *Time slice*, *q*). После окончания кванта времени *q* задача снимается с процессора и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению.

Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

Диспетчеризация без перераспределения процессорного времени, то есть *не вытесняющая многозадачность* (non-preemptive multitasking) – это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам, «по собственной инициативе», не отдаст управление диспетчеру задач для выбора из очереди другого, готового к выполнению процесса или треда. Дисциплины обслуживания FCFS, SJN, SRT относятся к невытесняющим.

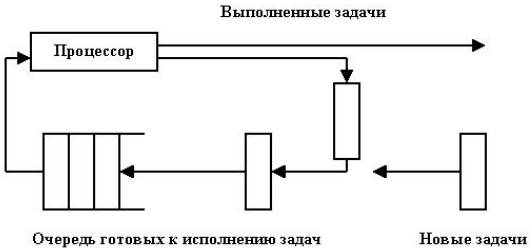

Рис. 1.7. Дисциплина диспетчеризации RR

Диспетчеризация с перераспределением процессорного времени между задачами, то есть *вытесняющая многозадачность* (preemptive multitasking) – это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается диспетчером задач, а не самой активной задачей. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе. При этом операционная система выполняет следующие функции:

- определяет момент снятия с выполнения текущей задачи,
- сохраняет ее контекст в дескрипторе задачи,
- выбирает из очереди готовых задач следующую и запускает ее на выполнение, предварительно загрузив ее контекст.

Дисциплина RR и многие другие, построенные на ее основе, относятся к вытесняющим.

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания, – это **гарантия обслуживания**. Дело в том, что при некоторых дисциплинах, например при использовании дисциплины абсолютных приоритетов, низкоприоритетные процессы оказываются обделенными многими ресурсами и, прежде всего, процессорным временем. Возникает реальная дискриминация низкоприоритетных задач, и некоторые такие процессы, имеющие к тому же большие потребности в ресурсах, могут очень длительное время откладываться или в конце концов вообще могут быть не выполнены.

Более жестким требованием к системе, чем просто гарантированное завершение процесса, является его гарантированное завершение *к* указанному моменту времени или за указанный интервал времени. Существуют различные дисциплины диспетчеризации, учитывающие жесткие временны́е ограничения.

Гарантировать обслуживание можно следующими тремя способами:

- выделять минимальную долю процессорного времени некоторому классу процессов, если по крайней мере один из них готов к исполнению. Например, можно отводить 20 % от каждых 10 мс процессам реального времени, 40 % от каждых 2 с – интерактивным процессам и 10 % от каждых 5 мин – фоновым процессам;
- выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению;
- выделять столько процессорного времени некоторому процессу, чтобы он мог выполнить свои вычисления к сроку.

Для сравнения алгоритмов диспетчеризации (оценки **качества диспетчеризации**) обычно используются следующие критерии.

- Использование (загрузка) центрального процессора (CPU utilization). В большинстве персональных систем средняя загрузка процессора не превышает 2…3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В серверах загрузка процессора колеблется в пределах 15…40 % для легко загруженного процессора и до 90…100 % – для сильно загруженного процессора.
- Пропускная способность (CPU throughput). Может измеряться количеством процессов, которые выполняются в единицу времени.
- Время оборота (turnaround time). Интервал от момента появления процесса во входной очереди до момента его завершения. Включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода/вывода.
- Время ожидания (waiting time). Суммарное время нахождения процесса в очереди готовых процессов.
- Время отклика (response time). Для интерактивных программ важным показателем является время отклика или время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу.

Очевидно, что простейшая стратегия краткосрочного планировщика должна быть направлена на максимизацию средних значений загруженности и пропускной способности, минимизацию времени ожидания и времени отклика.

Правильное планирование процессов сильно влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к уменьшению производительности системы.

- Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и перемещениями страниц виртуальной памяти, а также необходимостью обновления данных в кэше.
- Переключение на другой процесс в тот момент, когда текущий. процесс выполняет критическую секцию, а другие процессы активно ожидают входа в свою критическую секцию.

В случае использования мультипроцессорных систем применяются следующие методы повышения производительности системы:

- совместное планирование, при котором все потоки одного приложения (неблокированные) одновременно выбираются для выполнения процессорами и одновременно снимаются с них (для сокращения переключений контекста);
- планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие входа в критическую секцию задачи не выбираются до тех пор, пока вход в секцию не освободится;
- планирование с учетом так называемых «советов» программы (во время ее выполнения).

При выполнении программ, реализующих какие-либо задачи контроля и управления, может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных» задач (для которых истекает отпущенное для них время обработки). После выполнения этих задач их приоритет восстанавливается.

Поэтому в любой ОС реального времени имеются средства для изменения приоритета программ (*Dynamic priority variation*.)

Рассмотрим, например, как реализован механизм динамических приоритетов в ОС UNIX. Каждый процесс имеет два атрибута приоритета, с учетом которого и распределяется между исполняющимися задачами процессорное время:

- текущий приоритет, на основании которого происходит планирование;
- и заказанный относительный приоритет.

Текущий приоритет процесса варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи базовый приоритет меняется в диапазоне 0…65, для режима ядра – 66…95 (системный диапазон). Процессы, приоритеты которых лежат в диапазоне 96…127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение *приоритета сна,* выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшем это состояние. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет быстро завершить системный вызов, выполнение которого в свою очередь может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что в свою очередь вызовет переключение контекста.

В Windows NT каждый поток (тред) имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней ниже базового приоритета процесса, его породившего, до двух уровней выше этого приоритета, как показано на рис. 1.8.

Базовый приоритет процесса определяет, сколь сильно могут различаться приоритеты потоков процесса и как они соотносятся с приоритетами потоков других процессов. Поток наследует этот базовый приоритет и может изменять его так, чтобы он стал немного больше или немного меньше. В результате получается приоритет планирования, с которым поток и начинает исполняться. В процессе исполнения потока его приоритет может отклоняться от базового.

На рис. 1.8 показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя – зависит от вида работ, исполняемых потоком.

Снижая приоритет одного процесса и поднимая приоритет другого, подсистемы могут управлять относительной приоритетностью потоков внутри процесса.

Windows NT поддерживает 32 уровня приоритетов; потоки делятся на два класса: реального времени и переменного приоритета. Потоки реального времени, имеющие приоритеты от 16 до 31 – высокоприоритетные, из программ с критическим временем выполнения.

Большинство потоков в системе относятся к классу переменного приоритета с уровнями приоритета (номером очереди) от 1 до 15. Эти очереди используются потоками с переменным приоритетом (variable priority), так как диспетчер задач корректирует их приоритеты по мере выполнения задач для оптимизации отклика системы.

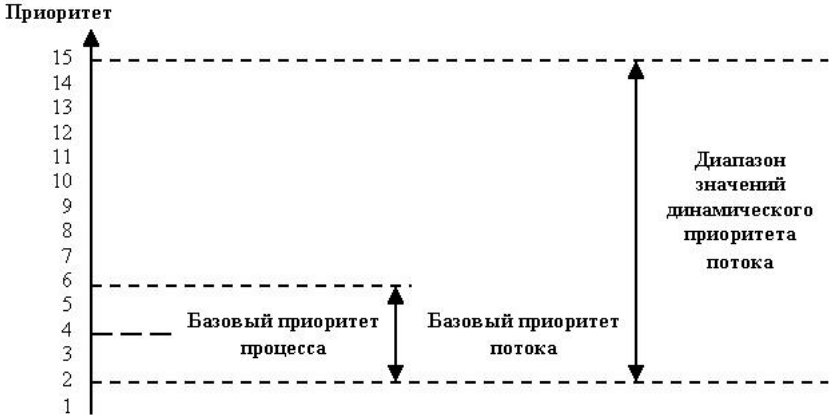

Рис. 1.8. Схема динамического изменения приоритетов в Windows NT

Диспетчер приостанавливает исполнение текущего потока после того, как тот израсходует свой квант времени. При этом если прерванный поток – это поток переменного приоритета, то диспетчер задач понижает его приоритет на единицу и перемещает в другую очередь. Таким образом, приоритет потока, выполняющего много вычислений, постепенно понижается (до значения его базового приоритета).

С другой стороны, диспетчер повышает приоритет потока после освобождения задачи (потока) из состояния ожидания. Величина этой добавки зависит от типа события, которого ожидал заблокированный поток. Так, например, поток, ожидавший ввода очередного байта с клавиатуры, получает бóльшую добавку к значению своего приоритета, чем процесс ввода/вывода, работавший с дисковым накопителем. Но в любом случае значение приоритета не может достигнуть 16.

1.8. ОСНОВНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ ОС

Рассмотрим кратко вопросы архитектуры ОС. Сделать это необходимо потому, что многие возможности и характеристики ОС определяются в значительной мере ее архитектурой. Среди множества принципов, которые используются при построении ОС, перечислим несколько наиболее важных.

1.8.1. Принцип модульности

Модуль – функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает возможность заменить его на другой при наличии заданных интерфейсов.

Особое значение при построении ОС имеют привилегированные, повторно входимые и реентерабельные (корректно выполняемые при рекурсивном вызове из прерывания) модули, так как они позволяют эффективнее использовать ресурсы вычислительной системы.

Принцип модульности отражает технологические и эксплуатационные свойства системы.

1.8.2. Принцип функциональной избирательности

В ОС выделяется некоторая часть важных модулей, которые должны постоянно находиться в оперативной памяти для более эффективной организации вычислительного процесса. Эту часть в ОС называют ядром. При формировании состава ядра требуется учитывать два противоречивых требования.

- В состав ядра должны войти наиболее часто используемые системные модули.
- Количество модулей должно быть таковым, чтобы объем памяти, занимаемый ядром, был не слишком большим.

В состав ядра, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно, средства по распределению таких основных ресурсов, как оперативная память и процессор.

Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые называются транзитными.

Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями. В качестве синонима к термину «транзитный» можно использовать термин «диск-резидентный».

1.8.3. Принцип генерируемости ОС

Это такой способ исходного представления центральной системной управляющей программы ОС (ее ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти), который позволял бы настраивать эту системную супервизорную часть, исходя из конкретной конфигурации конкретного вычислительного комплекса и круга решаемых задач.

Эта процедура проводится редко, перед протяженным периодом эксплуатации ОС. Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. Сгенерированная версия ОС представляет собой совокупность системных наборов модулей и данных.

Принцип генерируемости существенно упрощает настройку ОС на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости ОС можно столкнуться разве что только при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро ОС, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть – модулями, имеющими статус подгружаемых, транзитных.

1.8.4. Принцип функциональной избыточности

Этот принцип учитывает возможность проведения одной и той же работы различными средствами. В состав ОС может входить несколько типов мониторов (модулей супервизора, управляющих тем

или другим видом ресурса), различные средства организации коммуникаций между вычислительными процессами.

Наличие нескольких типов мониторов, нескольких систем управления файлами позволяет пользователям быстро и наиболее адекватно адаптировать ОС к определенной конфигурации вычислительной системы, обеспечить максимально эффективную загрузку технических средств при решении конкретного класса задач, получить максимальную производительность при решении заданного класса задач.

1.8.5. Принцип виртуализации

Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

Наиболее естественным и законченным проявлением концепции виртуальности является понятие виртуальной машины. По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией.

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками:

- единообразная по логике работы память (виртуальная) практически неограниченного объема. Среднее время доступа соизмеримо со значением этого параметра оперативной памяти;
- произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы;
- произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа.

Чем больше виртуальная машина, реализуемая средствами ОС на базе конкретной аппаратуры, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень виртуальности.

Одним из аспектов виртуализации является организация возможности выполнения в данной ОС приложений, которые разрабатывались для других ОС.

1.8.6. Принцип независимости программ от внешних устройств

Этот принцип заключается в том, что связь программ с конкретными устройствами производится не на уровне трансляции программы, а в период планирования ее исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется.

Принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не принесет каких-либо изменений в программу, если в системе реализован принцип независимости.

1.8.7. Принцип совместимости

Это способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы:

- совместимость на уровне команд процессора,
- совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для ПК типа IBM PC желательно выполнить на ПК типа Macintosh фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. Выходом в таких случаях является использование так называемых прикладных сред или эмуляторов.

Одним из средств обеспечения совместимости программных интерфейсов является соответствие стандартам POSIX, описанным ниже, при обсуждении программных интерфейсов. Использование стандарта POSIX позволяет создавать программы в стиле UNIX, которые впоследствии могут переноситься из одной системы в другую.

1.8.8. Принцип открытой и наращиваемой ОС

Открытая ОС доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему.

Наращиваемая (модифицируемая, развиваемая) ОС позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. Необходимо, чтобы можно было внести дополнения и изменения, и не нарушить целостность системы.

Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг – «серверов». Основная часть ОС остается неизменной и в то же время могут быть добавлены новые серверы или улучшены старые. Этот принцип иногда трактуют как расширяемость системы.

К открытым ОС прежде всего следует отнести UNIX-системы и, естественно, ОС Linux.

1.8.9. Принцип мобильности (переносимости)

Операционная система относительно легко должна переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и архитектуру вычислительной системы) одного типа на аппаратную платформу другого типа. Принцип переносимости очень близок принципу совместимости, но это не одно и то же.

Большая часть ОС должна быть написана на языке, который имеется на всех системах, на которые планируется в дальнейшем ее переносить. Это, прежде всего, означает, что ОС должна быть написана на языке высокого уровня, предпочтительно стандартизованном, например на языке С. Программа, написанная на ассемблере, не является в общем случае переносимой.

Важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами.

Если аппаратно-зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Например, можно спрятать аппаратно-зависимую структуру в программно задаваемые данные абстрактного типа.

Введение стандартов POSIX преследовало цель обеспечить переносимость создаваемого программного обеспечения.

1.8.10. Принцип обеспечения безопасности вычислений

Обеспечение безопасности при выполнении вычислений является желательным свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких, как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем. Во многих современных ОС гарантируется степень безопасности данных, соответствующая уровню С2 в системе стандартов США.

Безопасной считается система, которая «посредством специальных механизмов защиты контролирует доступ к информации таким обра-

зом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в стандартах, помечает низший уровень безопасности как D, а высший – как A.

В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

Основными свойствами, характерными для систем класса C, являются наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Класс (уровень) C делится на два подуровня: уровень C1, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников; и уровень C2. На уровне C2 должны присутствовать:

- средства секретного входа, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе;
- избирательный контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путем предоставления прав доступа пользователю или группе пользователей;
- средства учета и наблюдения (auditing), обеспечивающие возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы;
- защита памяти, заключающаяся в том, что память инициализируется перед тем, как повторно используется.

Системы уровня B основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.

Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального, математически обоснованного доказательства соответствия системы требованиям безопасности. A-уровень безопасности занимает своими управляющими механизмами до 90 % процессорного времени.

Более безопасные системы существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например, для ОС Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога В уровня – только около ста.

1.9. МИКРОЯДЕРНЫЕ ОС

Микроядро – это минимальная стержневая часть операционной системы, служащая основой модульных и переносимых расширений. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, и ряд других функций.

Остальные функции обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой и осуществляющие взаимодействие посредством передачи сообщений.

Микроядро является маленьким модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим остальную часть операционной системы, рассматриваемую как набор серверных приложений.

Микроядро включает только те функции, которые требуются для определения набора абстрактных сред обработки для прикладных программ и для организации совместной работы приложений в обеспечении сервисов и в действии с клиентами и серверами. В результате микроядро обеспечивает только пять различных типов сервисов:

- управление виртуальной памятью;
- упракление процессами и потоками;
- межпроцессные коммуникации (IPC – inter-process communication);
- управление поддержкой ввода/вывода и прерываниями;
- сервисы набора хоста (host – компьютер, имеющий IP-адрес) и процессора.

Наиболее ярким представителем микроядерных ОС является ОС реального времени QNX. Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня. Микроядро

может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486. Разные версии этой ОС имели и различные объемы ядер – от 8 до 46 Кбайт.

Чтобы построить минимальную систему QNX, требуется добавить к микроядру менеджер процессов, который создает процессы, управляет процессами и памятью процессов. Чтобы ОС QNX была применима не только во встроенных и бездисковых системах, нужно добавить файловую систему и менеджер устройств.

1.10. МОНОЛИТНЫЕ ОС

В монолитной ОС, несмотря на ее возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных ОС требуют очень хорошего знания всей архитектуры ОС и больших усилий.

При поддержке монолитных ОС возникает ряд проблем, связанных с тем, что все функции макроядра работают в едином адресном пространстве:

- опасность возникновения конфликта между частями ядра;
- сложность подключения к ядру новых драйверов.

Преимущество микроядерной архитектуры перед монолитной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Существует подход к проектированию ОС, известный как «клиент-серверная» технология, который позволяет в большей мере и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования ОС.

Модель клиент–сервер предполагает наличие программного компонента, являющегося потребителем какого-либо сервиса – клиента, и компонента, служащего поставщиком этого сервиса – сервера. Взаимодействие клиента и сервера стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, может быть, разными разработчиками. При этом главным требованием является использование единообразного интерфейса. Инициатором обмена является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса.

Использование технологии клиент–сервер – еще не гарантия того, что ОС станет микроядерной. В качестве подтверждения можно привести пример с ОС Windows NT, которая построена на идеологии клиент-сервер, но которую тем не менее трудно назвать микроядерной.

1.11. ПРИНЦИПЫ ПОСТРОЕНИЯ ИНТЕРФЕЙСОВ ОС

ОС всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем следует понимать специальные интерфейсы системного и прикладного программирования, предназначенные для выполнения следующих задач.

- Управление процессами, которое включает в себя следующий набор основных функций:
  - запуск, приостанов и снятие задачи с выполнения;
  - задание или изменение приоритета задачи;
  - взаимодействие задач между собой (механизмы сигналов, семафоры, очереди, конвейеры, почтовые ящики);
  - RPC (remote procedure call) – удаленный вызов подпрограмм.
- Управление памятью:
  - запрос на выделение блока памяти;
  - освобождение памяти;
  - изменение параметров блока памяти (память может быть заблокирована процессом либо предоставлена в общий доступ);
  - отображение файлов на память (имеется не во всех системах).
- Управление вводом/выводом:
  - запрос на управление виртуальными устройствами (управление вводом/выводом является привилегированной функцией ОС);
  - файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, собранных в файлы).

Пользовательский интерфейс операционной системы реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд.

Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо обращается к другим модулям ОС, используя механизм API (Application Program Interface, интерфейс прикладного программирования). В случае графических интерфейсов (GUI, Graphic User Interface) указание курсором на объекты и щелчок (клик) или двойной щелчок по соответствующим клавишам приводит к каким-либо действиям – запуску программы, ассоциированной с указываемым объектом, выбору и/или активизации пунктов меню и т. д. Такая интерфейсная подсистема транслирует «команды» пользователя в обращения к ОС. Управление GUI – частный случай задачи управления вводом/выводом, не являющийся частью ядра операционной системы, хотя в ряде случаев разработчики ОС относят функции GUI к основному системному API.

1.11.1. Интерфейс прикладного программирования

Необходимо разделить термин API на следующие направления:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL. RTL (run time library) – библиотека времени выполнения; она включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае RTL включает в себя не только модули из системы программирования, но и модули самой ОС;
- API прикладных и системных программ, входящих в поставку операционной системы;
- прочие API.

Интерфейс прикладного программирования предназначен для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС. Представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы *с целевой вычислительной системой.* Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа.

API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются ОС, архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

- реализация на уровне ОС;
- реализация на уровне системы программирования;
- реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

- эффективность выполнения функций API – включает в себя скорость выполнения функций и объем вычислительных ресурсов, потребных для их выполнения;
- широта предоставляемых возможностей;
- зависимость прикладной программы от архитектуры целевой вычислительной системы.

1.11.2. Реализация функций API на уровне ОС

За их выполнение ответственность несет ОС. Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек, разработанных для данной ОС. Система программирования ответственна за то, чтобы организовать интерфейс для вызова этого кода. Результирующая программа обращается непосредственно к ОС, поэтому достигается наибольшая эффективность выполнения функций API по сравнению с другими вариантами реализации API.

Недостаток организации API по такой схеме – полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на другой вычислительной системе даже после того, как ее объектный код будет полностью перестроен. Зачастую система программирования не сможет выполнить перестроение исходного кода дли новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную ОС, в новой архитектуре просто отсутствуют. Для переноса прикладной программы с одной вычислительной системы на другую требуется изменение исходного кода программы.

Пример API такого рода – набор функций, предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows

API) [2]. Даже внутри этого корпоративного API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными ОС типа Windows.

Еще один пример такого API – набор сервисных функций ОС типа MS-DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

1.11.3. Реализация функций API на уровне системы программирования

Функции API предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени исполнения – RTL (run time library). Система программирования предоставляет пользователю библиотеку соответствующего языка программирования и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Эффективность функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям ОС. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования должна все равно выполнять обращения к функциям ОС. Наличие всех необходимых вызовов и обращений к функциям ОС в объектном коде RTL обеспечивает система программирования.

Рассмотрим вызов в программе на языке С для ОС Windows функции по запросу 256 байт памяти

```
unsigned char * ptr = malloc (256);
```

Из кода пользовательской программы будет осуществлен вызов библиотечной функции **`malloc`**, код которой расположен в RTL. Библиотека времени выполнения в данном случае реализует вызов malloc уже как вызов системной функции API **`HeapAlloc:`**

```
LPVOID HeapAlloc;
HANDLE hHeap; /* указатель на блок */
DWORD dwFlags; /* свойства блока */
DWORD dwBytes; /* размер блока */
```

Параметры выделяемого блока памяти в таком случае задаются системой программирования, и пользователь лишен возможности задавать их напрямую. Но можно использовать функции API прямо в тексте программы, подгрузив системную библиотеку Kernel32.dll:

```
unsigned char * ptr = (LPVOID) HeapAlloc( GetProcess-
Heap(), 0, 256);
```

Переносимость исходного кода программы при использовании RTL будет самой высокой, поскольку синтаксис и семантика всех функций строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка, а не от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой платформе достаточно заново построить код результирующей программы с помощью соответствующей системы программирования. Для каждой платформы будет требоваться свой код RTL языка программирования.

1.11.4. Реализация функций API с помощью внешних библиотек

Внешние библиотеки предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

Эффективность выполнения – самая низкая, ибо внешняя библиотека обращается как к функциям ОС, так и к функциям RTL языка программирования.

Требование к переносимости исходного кода только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX, доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым.

1.11.5. Платформенно-независимый интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments) – платформенно независимый системный интерфейс для компьютерного окружения. Это стандарт IEEE, описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инст-

рументарии. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

Этот стандарт подробно описывает VMS (virtual memory system, систему виртуальной памяти), многозадачность (MPE, multiprocess executing) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов, именуемых POSIX.1 – POSIX.12.

В табл. 1 приведены основные направления, описываемые данными стандартами. Следует отметить, что POSIX.1 предполагает язык С как основной язык описания системных функций API.

Таблица 1.1

Семейство стандартов POSIX

Стандарт	Стандарт ISO	Краткое описание
POSIX.0	Нет	Введение в стандарт открытых систем. Не является стандартом, а представляет собой рекомендации и краткий обзор технологий
posix.1	Да	Системный API (язык С)
POSIX.2	Нет	Оболочки и утилиты (одобренные IEEE)
POSIX.3	Нет	Тестирование и верификация
POSIX.4	Нет	Задачи реального времени и нити
POSIX.5	Да	Использование языка ADA применительно к стандарту POSIX.1
POSIX.6	Нет	Системная безопасность
POSIX.7	Нет	Администрирование системы
POSIX.8	Нет	Сети. «Прозрачный» доступ к файлам. Абстрактные сетевые интерфейсы, не зависящие от физических протоколов. RPC. Связь системы с протоколо-зависимыми приложениями
POSIX.9	Да	Использование языка FORTRAN применительно к стандарту POSIX.1
POSIX.10	Нет	Super-computing Application Environment Profile (AEP)
POSIX.11	Нет	Обработка транзакций AEP
POSIX.12	Нет	Графический интерфейс пользователя

На рис. 1.9 изображена типовая схема реализации строго соответствующего POSIX приложения.

Видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка C, в которой возможно использование лишь 110 различных функций, также описанных стандартом POSIX.1.

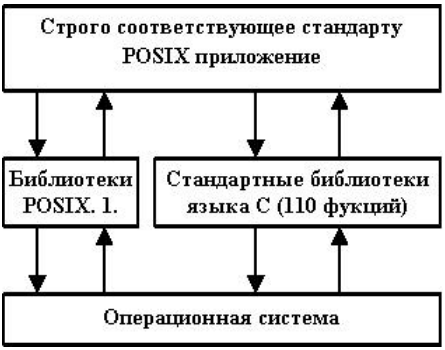

Рис. 1.9. Приложения, строго соответствующие стандарту POSIX

Реализации POSIX API на уровне операционной системы различны. Если UNIX-системы изначально соответствуют спецификациям IEEE Standard 1003.1–1990, то WinAPI не является POSIX-совместимым. Для его поддержки в MS Windows NT введен специальный модуль поддержки POSIX API, работающий на уровне привилегий пользовательских процессов. Он обеспечивает конвертацию и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Приложения, написанные с использованием WinAPI, могут передавать информацию POSIX-приложениям через стандартные механизмы потоков ввода/вывода (stdin, stdout).

1.11.6. Пример программирования в различных API ОС

Для демонстрации принципиальных различий API наиболее популярных современных операционных систем для ПК рассмотрим простейший пример, в котором реализуется следующая задача.

Постановка задачи: необходимо подсчитать количество пробелов в текстовых файлах, имена которых указываются в командной строке. При выполнении родительской программы для каждого перечисленного в командной строке файла создается свой (дочерний) процесс, который параллельно с другими процессами производит подсчет пробелов в «своем» файле. Результатом работы программ будет являться список файлов с подсчитанным количеством пробелов для каждого.

Рассмотрим два варианта программы, решающей эту задачу, – для Windows (file.cpp, spaces.cpp) и для Linux (file.c, spaces.c). Для каждой ОС имеются тексты дочерних и родительских программ. Тексты можно скачать с сайта [3] в папке «API ОС».

2. МНОГОЗАДАЧНОЕ И МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

2.1. ПРОЦЕССЫ И ПОТОКИ В WINDOWS

Процесс есть объект, обладающий собственным независимым виртуальным адресным пространством, в котором могут размещаться код и данные, защищенные от других процессов. Внутри каждого процесса могут независимо выполняться один или несколько *потоков*. Поток может создавать новые потоки и новые независимые процессы, управлять взаимодействием объектов между собой и их синхронизацией.

Внутри каждого процесса могут выполняться один или несколько потоков, и именно поток является базовой единицей выполнения в Windows. Выполнение потоков планируется системой на основе обычных факторов: наличие таких ресурсов, как CPU и физическая память, приоритеты, равнодоступность ресурсов и т. д. Начиная с Windows NT4 поддерживается симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), позволяющая распределять выполнение потоков между отдельными процессорами.

Каждому процессу принадлежат следующие ресурсы.

- Один или несколько потоков.
- Виртуальное адресное пространство, отличное от адресных пространств других процессов.
- Один или несколько сегментов кода, включая код DLL.
- Один или несколько сегментов данных, содержащих глобальные переменные.
- Строки, содержащие информацию об окружении, например, информацию о текущем пути доступа к файлам.
- Область нераспределенной памяти (куча, heap) процесса.
- Различного рода ресурсы, например, дескрипторы открытых файлов.

Поток разделяет вместе с процессом код, глобальные переменные, строки окружения и другие ресурсы. Каждый поток планируется независимо от других и располагает следующими элементами.

- Стек, используемый для вызова процедур, прерываний и обработчиков исключений, а также хранения автоматических переменных.

- Локальные области хранения потока (Thread Local Storage, TLS) – массивы указателей, используя которые, каждый поток может создавать собственную уникальную информационную среду.
- Аргумент в стеке, получаемый от создающего потока, который обычно является уникальным для каждого потока.
- Структура контекста, поддерживаемая ядром системы и содержащая значения машинных регистров.

На рис. 2.1 показан процесс с несколькими потоками.

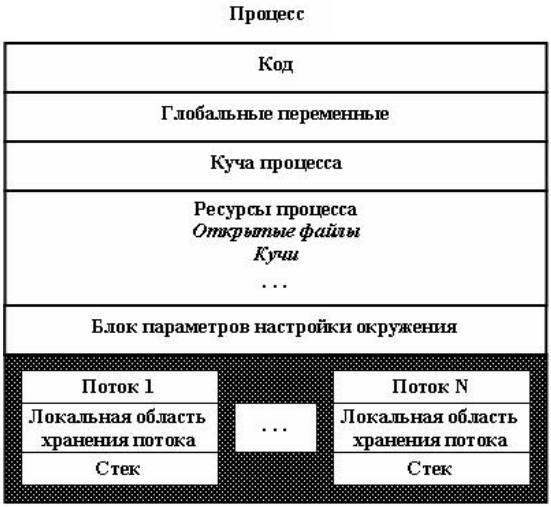

Рис. 2.1. Процесс и его потоки

2.2. МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

2.2.1. Создание процесса

Функция **`CreateProcess`** создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы. Для большинства параметров можно использовать значения, заданные по умолчанию (NULL).

```
BOOL CreateProcess(lpApplicationName, LPTSTR lpCommand-
Line, LPSECURITY_ATTRIBUTES lpsaProcess, LPSECURITY_ATTRI-
BUTES lpsaThread, BOOL bInheritHandles, DWORD dwCreation-
Flags, LPVOID lpEnvironment, LPCTSTR lpCurDir, LPSTAR-
TUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcInfo);
```

Функция возвращает два дескриптора, по одному для процесса и потока, передавая их в структуре типа **PPROCESS_INFORMATION**. Эти дескрипторы относятся к создаваемому функцией **CreateProcess** новому процессу и его *основному* потоку. Во избежание утечки ресурсов в процессе работы необходимо закрывать оба дескриптора, когда они больше не нужны. Возвращаемое значение: в случае успешного создания процесса и потока – **TRUE**, иначе – **FALSE**.

Параметры

lpApplicationName и **lpCommandLine** (последний указатель имеет тип **LPTSTR**, а не **LPCTSTR**) – используются вместе для указания исполняемой программы и аргументов командной строки. При этом действуют следующие правила.

- Указатель **lpApplicationName**, если его значение не равно NULL, указывает на строку, содержащую имя файла исполняемого модуля. Если имя модуля содержит пробелы, его заключают в кавычки.
- Если значение указателя **lpApplicationName** равно NULL, то имя модуля определяется первой из лексем в параметре **lpCommandLine**.

Обычно задается только параметр **lpCommandLine**, в то время как параметр **lpApplicationName** полагается равным NULL. Новый процесс может получить командную строку посредством обычного **argv**-механизма или путем вызова функции **GetCommandLine** для получения командной строки в виде одиночной строки символов.

lpsaProcess и **lpsaThread** – указатели на структуры атрибутов защиты процесса и потока. Значениям NULL соответствует использование атрибутов защиты, заданных по умолчанию.

bInheritHandles – режим наследования открытых дескрипторов файлов, и так далее, из вызывающего процесса. Для наследования значение параметра устанавливают равным **TRUE**, иначе **FALSE**.

dwCreationFlags – может объединять в себе несколько флаговых значений, включая следующие.

- **CREATESUSPENDED** – указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функции **ResumeThread**.

- **DETACHED_PROCESS** и **CREATE_NEW_CONSOLE** – взаимоисключающие флаги: первый означает создание нового процесса, у которого консоль отсутствует, а второй – процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса.
- **Create_New_Process_Group** – указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C).
- флаги управления приоритетами потоков нового процесса. Можно использовать приоритет родительского процесса (устанавливается по умолчанию) или указывать значение **NORMAL_PRIORITY_CLASS**.

lpEnvironment – блок параметров настройки окружения нового процесса. Если задано значение NULL, то новый процесс будет использовать значения параметров окружения родительского процесса.

lpCurDir – указатель на строку, содержащую путь к текущему каталогу нового процесса. Если задано значение NULL, то будет использоваться рабочий каталог родительского процесса.

lpStartupInfo – указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса. Соответствующую информацию из родительского процесса можно получить при помощи функции **GetStartupInfo**. Можно обнулить структуру **STARTUPINFO** перед вызовом функций **CreateProcess**.

lpProcInfo – указатель на структуру, в которую помещаются значения дескрипторов и идентификаторов процесса и потока.

Структура **PROCESS_INFORMATION** имеет следующий вид:

```
typedef struct PROCESS_INFORMATION {
HANDLE hProcess; HANDLE hThread;
DWORD dwProcessId; DWORD dwThreadId;}
PROCESS_INFORMATION;
```

Процессам и потокам нужны и дескрипторы, и идентификаторы, поскольку одним функциям управления требуются идентификаторы, а другим – дескрипторы. Дескрипторы процессов и потоков должны закрываться после того как необходимость в них отпала.

2.2.2. Завершение и прекращение выполнения процесса

После того как процесс завершил свою работу, он может вызвать функцию **ExitProcess**, указав в качестве параметра код завершения:

```
VOID ExitProcess (UINT uExitCode)
```

Эта функция не осуществляет возврата. Она завершает вызывающий процесс и все его потоки. Выполнение оператора **return** в основной программе с использованием кода возврата равносильно вызову функции **ExitProcess**, в котором этот код возврата указан в качестве кода завершения. Другой процесс может определить код завершения, вызвав функцию **GetExitCodeProcess**:

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExit-
Code)
```

Процесс, идентифицируемый дескриптором **hProcess**, должен обладать правами доступа **PROCESS_QUERY_INFORMATION** (см. описание функции **OpenProcess**). **lpExitCode** указывает на переменную типа **DWORD**, вторая принимает значение кода завершения. Одним из ее возможных значений является **STILL_ACTIVE**, означающее, что данный процесс еще не завершился.

Один процесс может прекратить выполнение другого процесса, если у дескриптора завершаемого процесса имеются права доступа **PROCESS_TERMINATE**. При вызове функции завершения процесса указывается код завершения:

```
BOOL TerminateProcess (HANDLE hProcess, UINT uExitCode).
```

Прежде чем завершить выполнение процесса, все ресурсы, которые он мог разделять с другими процессами, должны быть освобождены.

2.2.3. Ожидание завершения процесса

Простейшим методом синхронизации с другим процессом является ожидание его завершения. Представленные ниже стандартные функции ожидания Windows обладают рядом интересных свойств.

- Функции ожидания могут работать с самыми различными типами объектов (процессами и потоками).
- Функции могут ожидать завершения одного процесса, первого из нескольких указанных, или всех процессов, образующих группу.
- Есть возможность устанавливать конечный интервал ожидания.

```
DWORD WaitForSingleObject (HANDLE hObject, DWORD dwMil-
liseconds)
DWORD WaitForMultipleObjects (DWORD nCount, CONST HANDLE
*lpHandles, BOOL fWaitAll, DWORD dwMilliseconds)
```

Возвращаемое значение указывает причину завершения ожидания или, в случае ошибки, равно 0xFFFFFFFF (для получения более подробной информации используйте функцию **GetLastError**).

В аргументах этих функций указывается либо дескриптор одиночного процесса (**hObject**), либо дескрипторы ряда отдельных объектов, хранящиеся в массиве, на который указывает указатель **lpHandles**. Значение параметра **nCount**, определяющего размер массива, не должно превышать 64 (определено в файле WINNT.H).

dwMilliseconds – число миллисекунд интервала ожидания. Если число равно 0, то возврат из функции осуществляется сразу же, что полезно при опросе состояния процессов. Если же значение параметра равно **INFINITE**, то ожидание длится до завершения процесса.

fWaitAll – параметр, указывающий (если его значение равно **TRUE**) на необходимость ожидания завершения всех процессов.

Возможные возвращаемые значения функций:

- **WAIT_OBJECT_0** – указанный объект завершен или остановлен (в случае функции **WaitForSingleObject**) или что одновременно все **nCount** объектов завершены или остановлены (в случае функции **WaitForMultipleObject**, когда параметр **fWaitAll** равен **TRUE**);
- **WAIT_OBJECT_0+n**, где 0 < n < **nCount** – вычтя значение **WAIT_OBJECT_0** из возвращенного значения, можно определить, выполнение какого процесса завершилось, если ожидается завершение выполнения любого из группы процессов. Если завершено несколько объектов, возвращается наименьшее из возможных значений;
- **WAIT_TIMEOUT** – указывает на то, что в течение отведенного периода ожидания объект (объекты) не завершены;
- **WAIT_FAILED** – неудачное завершение функции, вызванное, например, тем, что у дескриптора отсутствовали права доступа.

Программа spaces.cpp (см. стр. 43) иллюстрирует применение методики, обеспечивающей взаимодействие процессов.

2.3. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ИНФОРМАЦИИ ПРОЦЕССАМИ

Основными механизмами Windows, реализующими межпроцессное взаимодействие (Interprocess Communication, IPC), являются анонимные и именованные каналы, доступ к которым осуществляется с помощью функций **ReadFile** и **WriteFile**. Анонимные

каналы являются символьными и работают в полудуплексном режиме. Именованные каналы являются дуплексными, ориентированы на обмен сообщениями и обеспечивают взаимодействие через сеть. Один именованный канал может иметь несколько открытых дескрипторов. Поэтому именованные каналы пригодны для создания клиент-серверных систем.

Дополнительные механизмы IPC включают отображаемые файлы, удаленные вызовы процедур, отправку сообщений через почтовые ящики.

2.3.1. Анонимные каналы

Анонимные каналы Windows обеспечивают однонаправленное (полудуплексное) посимвольное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle). Функция, с помощью которой создаются анонимные каналы, имеет следующий прототип:

```
BOOL CreatePipe (PHANDLE phRead, PHANDLE phWrite, LPSE-
CURITY_ATTRIBUTES lpsa, DWORD cbPipe)
```

Дескрипторы каналов часто бывают наследуемыми. Значение параметра cbPipe указывает размер канала в байтах, причем значению 0 соответствует размер канала по умолчанию.

Для организации IPC через канал должен существовать еще один процесс, для которого требуется один из дескрипторов канала. Если родительскому процессу, вызвавшему функцию **`CreatePipe`**, необходимо передать данные дочернему процессу, то он передает ему дескриптор чтения (**`phRead`**), устанавливая дескриптор стандартного ввода в структуре **`STARTUPINFO`** для дочерней процедуры равным **`*phRead`**.

Чтение из канала блокируется, если канал пуст. В противном случае в процессе чтения будет воспринято столько байтов, сколько имеется в канале, вплоть до количества, указанного при вызове функции **`ReadFile`**. Операция записи в заполненный канал, которая выполняется с использованием буфера в памяти, также будет блокирована.

В программе pipe.c представлен родительский процесс, который создает два процесса из командной строки и соединяет их каналом, осуществляя перенаправление стандартного ввода/вывода дочерних процессов.

Дескрипторы каналов должны закрываться при первой же возможности. Родительский процесс должен закрыть дескриптор устройства

стандартного вывода сразу же после создания первого дочернего процесса, чтобы второй процесс мог распознать метку конца файла, когда завершится выполнение первого процесса. В случае существования открытого дескриптора первого процесса второй процесс не смог бы завершиться, поскольку система обозначила бы конец файла.

Рис. 2.2 схематично представляет выполнение следующей команды:

```
> pipe Program1 аргументы = Program2 аргументы
```

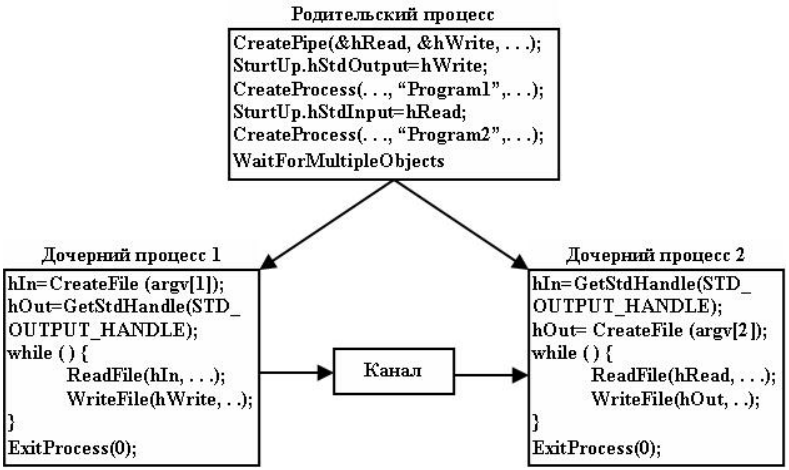

Рис. 2.2. Процесс и его потоки

При использовании средств командного процессора UNIX или Windows соответствующая команда имела бы следующий вид:

```
> Program1 аргументы | Program2 аргументы
```

В программу pipe.c передаются две команды, разделенные символом =, обозначающим канал. Использование символа вертикальной черты (|) привело бы к возникновению конфликта с системным командным процессором. Программа pipe.c доступна на сайте в папке «WinAPI».

2.3.2. Именованные каналы

Именованные каналы предлагают следующие возможности, которые делают их полезными в качестве универсального механизма реализации приложений на основе IPC.

- Именованные каналы ориентированы на обмен сообщениями, поэтому процесс, выполняющий чтение, может считывать сообщения переменной длины именно в том виде, в каком они были посланы процессом, выполняющим запись.
- Именованные каналы являются двунаправленными, что позволяет осуществлять обмен сообщениями между двумя процессами посредством единственного канала.
- Допускается существование нескольких независимых экземпляров канала, имеющих одинаковые имена.
- Каждая из систем, подключенных к сети, может обратиться к каналу, используя его имя. Взаимодействие посредством именованного канала осуществляется одинаковым образом для процессов, выполняющихся как на одной и той же, так и на разных машинах.
- Имеется несколько вспомогательных и связных функций, упрощающих обслуживание взаимодействия «запрос/ответ» и клиент-серверных соединений.

Когда требуется, чтобы канал связи был двунаправленным, ориентированным на обмен сообщениями или доступным для нескольких клиентских процессов, следует применять именованные каналы.

Функция **`CreateNamedPipe`** создает первый экземпляр именованного канала и возвращает дескриптор. При вызове этой функции указывается также максимально допустимое количество экземпляров каналов, а следовательно, и количество клиентов, одновременная поддержка которых может быть обеспечена.

Как правило, создающий процесс рассматривается в качестве сервера. Клиентские процессы, которые могут выполняться и на других системах, открывают канал с помощью функции **`CreateFile`**.

На рис. 2.3 представлены отношения «клиент-сервер», а также псевдокод, отражающий одну из возможных схем применения именованных каналов. Сервер создает множество экземпляров одного и того же канала, каждый из которых обеспечивает поддержку одного клиента. Для каждого экземпляра именованного канала сервер создает поток, так что каждого клиента обслуживает выделенный поток через экземпляр именованного канала. Возможно создание процесса-сервера.

Серверами именованных каналов могут быть только системы на основе Windows NT (4.0 и выше); системы на базе Windows 9x могут выступать только в роли клиентов.

Прототип функции **CreateNamedPipe**:

```
HANDLE CreateNamedPipe (LPCTSTR lpName, DWORD dwOpen-
Mode, DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOut-
BufferSize, DWORD nInBufferSize, DWORD nDefaultTimeOut,
LPSECURITY ATTRIBUTES lpSecurityAttributes);
```

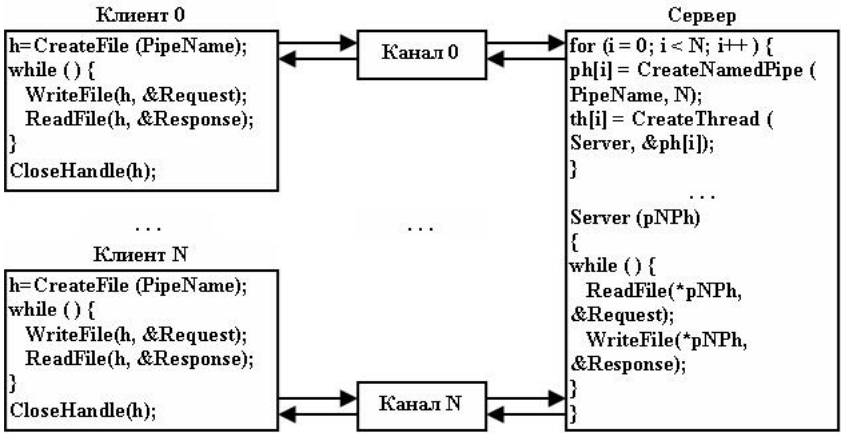

Рис. 2.3. Взаимодействие клиентов с сервером через именованные каналы

Параметры

lpName – указатель на имя канала, который должен иметь форму **\\.\pipe\ [path]pipename.** Точка (.) обозначает локальный компьютер; то есть создать канал на удаленном компьютере невозможно.

dwOpenMode – указывает один из следующих флагов:

- **PIPE_ACCESS_DUPLEX** – этот флаг эквивалентен комбинации значений **GENERIC_READ** и **GENERIC_WRITE**;
- **PIPE_ACCESS_INBOUND** – данные могут передаваться только в направлении от клиента к серверу; эквивалентно **GENERIC_READ**;
- **PIPE_ACCESS_OUTBOUND** – этот флаг эквивалентен **GENERIC_WRITE**.

dwPipeMode – имеются три пары взаимоисключающих значений этого параметра: ориентирована ли запись на работу с сообщениями или байтами, ориентировано ли чтение на работу с сообщениями или байтами, и блокируются ли операции чтения:

- **PIPE_TYPE_BYTE** и **PIPE_TYPE_MESSAGE** – указывают, соответственно, должны ли данные записываться в канал как поток байтов или как

сообщения. Для всех экземпляров каналов с одинаковыми именами следует использовать одно и то же значение;

- **PIPE_READMODE_BYTE** и **PIPE_READMODE_MESSAGE** – указывают, соответственно, должны ли данные считываться как поток байтов или как сообщения. Значение **PIPE_READMODE_MESSAGE** требует использования значения **PIPE_TYPE_MESSAGE**;
- **PIPE_WAIT** и **PIPE_NOWAIT** – определяют, соответственно, будет или не будет блокироваться операция **ReadFile**. Следует использовать значение **PIPE_WAIT**, поскольку для обеспечения асинхронного ввода/вывода существуют лучшие способы.

nMaxInstances – определяет количество экземпляров каналов. Как показано на рис. 2.3, при каждом вызове функции **CreateNamedPipe** для данного канала должно использоваться одно и то же значение. Чтобы предоставить ОС возможность самостоятельно определить значение этого параметра на основании доступных системных ресурсов, следует указать значение **PIPE_UNLIMITED_INSTANCES**.

nOutBufferSize и **nInBufferSize** – позволяют указать размеры (в байтах) выходного и входного буферов именованных каналов. Чтобы использовать размеры буферов по умолчанию, укажите значение 0.

nDefaultTimeOut – длительность интервала ожидания по умолчанию (в миллисекундах) для функции **WaitNamedPipe**. Эта ситуация, в которой функция, создающая объект, устанавливает интервал ожидания для родственной функции, является уникальной.

В случае ошибки возвращается значение **INVALID_HANDLE_VALUE**, поскольку дескрипторы каналов аналогичны дескрипторам файлов. При попытке создания именованного канала под управлением Windows 9x, которая не может выступать в качестве сервера именованных каналов, возвращаемым значением будет NULL.

lpSecurityAttributes – имеет тот же смысл, что и в случае любой функции, создающей объект.

При первом вызове функции **CreateNamedPipe** происходит создание самого именованного канала. Закрытие последнего открытого дескриптора экземпляра именованного канала приводит к уничтожению этого экземпляра. Уничтожение последнего экземпляра именованного канала приводит к уничтожению самого канала, в результате чего имя канала становится вновь доступным для повторного использования.

Для подключения клиента к именованному каналу применяется функция **CreateFile**, при вызове которой указывается имя именован-

ного канала. Если клиент и сервер выполняются на одном компьютере, то для указания имени канала используется форма: **\\.\pipe\[path]pipename.** Если сервер находится на другом компьютере, для указания имени канала используется форма: **\\servername\pipe\[path]pipename.** Использование точки (.) вместо имени локального компьютера в случае, когда сервер является локальным, позволяет значительно сократить время подключения.

Предусмотрены две функции, позволяющие получать информацию о состоянии каналов, и еще одна функция, позволяющая устанавливать данные состояния канала:

- **GetNamedPipeHandleState** – возвращает для заданного открытого дескриптора информацию относительно того, работает ли канал в блокируемом или неблокируемом режиме, ориентирован ли он на работу с сообщениями или байтами, каково количество экземпляров канала и тому подобное.
- **SetNamedPipeHandleState** – позволяет программе устанавливать атрибуты состояния. Параметр режима (**NpMode**) передается не по значению, а по адресу, что может стать причиной недоразумений.
- **GetNamedPipeInfo** – определяет, принадлежит ли дескриптор экземпляру клиента или сервера, размеры буферов и прочее.

После создания именованного канала сервер может ожидать подключения клиента (осуществляемого с помощью функции **CreateFile**), используя для этого функцию **ConnectNamedPipe**, которая является серверной функцией лишь в случае Windows NT:

```
Bool ConnectNamedPipe (HANDLE hNamedPipe, LPOVERLAPPED lpOverlapped);
```

Если параметр **lpOverlapped** установлен в NULL, то функция **ConnectNamedPipe** осуществляет возврат сразу же после установления соединения с клиентом. В случае успешного выполнения функции возвращаемым значением является **TRUE**. Если же подключение клиента происходит между вызовами сервером функций **CreateNamedPipe** и **ConnectNamedPipe**, то возвращается значение **FALSE**, а функция **GetLastError** вернет значение **ERROR_PIPE_CONNECTED**.

После возврата из функции **ConnectNamedPipe** сервер может выполнять чтение запросов с помощью функции **ReadFile** и запись ответов посредством функции **WriteFile**. Наконец, сервер должен вызвать функцию **DisconnectNamedPipe**, чтобы освободить дескриптор экземпляра канала для соединения с другим клиентом.

Последняя функция, **WaitNamedPipe**, используется клиентами для синхронизации соединений с сервером. Функция осуществляет успешный возврат, когда на сервере имеется незавершенный вызов функции **ConnectNamedPipe**, указывающий на наличие доступного экземпляра именованного канала. Используя **WaitNamedPipe**, клиент имеет возможность убедиться в том, что сервер готов к образованию соединения, после чего может вызвать функцию **CreateFile**. Вместе с тем вызов клиентом функции **CreateFile** может завершиться ошибкой, если в это же время другой клиент открывает экземпляр именованного канала или дескриптор экземпляра закрывается сервером. При этом неудачного завершения вызванной сервером функции **ConnectNamedPipe** не произойдет. Заметьте, что для функции **WaitNamedPipe** предусмотрен интервал ожидания, который, если он указан, отменяет значение интервала ожидания, заданного при вызове серверной функции **CreateNamedPipe**.

Последовательность операций, выполняемых сервером: сервер создает соединение с клиентом, взаимодействует с клиентом до тех пор, пока тот не разорвет единение (вынуждая функцию **ReadFile** вернуть значение **FALSE**), разрывает соединение на стороне сервера, образует соединение с другим клиентом:

```
hNp = CreateNamedPipe ("\\\\. \\pipe\\my_pipe",    ...);
while   ( /*Цикл до завершения работы сервера.*/)
{
ConnectNamedPipe (hNp, NULL);
while (ReadFile (hNp, Request, ...) {
WriteFile (hNp, Response, ...);
      }
DisconnectNamedPipe (hNp);
}
CloseHandle (hNp);
```

Последовательность операций, выполняемых клиентом:

```
 WaitNamedPipe("\\\\ServerName\\pipe\\my_pipe",
     NMPWAIT_WAIT_FOREVER);
 hNp = CreateFile("\\\\ServerName\\pipe\\my_pipe",...);
 while (/*Цикл, пока не прекратятся запросы.*/
 {
 WriteFile (hNp, Request,...);
 ReadFile  (hNp, Response);
 }
 CloseHandle (hNp);
```

Клиентский вызов функции **WaitNamedPipe** завершится ошибкой, если именованный канал к этому моменту еще не был создан сервером. В редких случаях вызов **CreateFile** может быть выполнен еще до того, как сервер вызовет функцию **ConnectNamedPipe**. В этом случае функция **ConnectNamedPipe** вернет серверу значение **FALSE**, однако взаимодействие посредством именованного канала по-прежнему будет функционировать надлежащим образом. Экземпляр именованного канала – глобальный ресурс, поэтому, когда клиент разрывает соединение с сервером, к нему может подключиться другой клиент.

Можно определить, имеются ли в канале фактические сообщения, используя для этого функцию **PeekNamedPipe**. Это средство может быть использовано для опроса именованного канала, определения размера сообщения, чтобы распределить память для буфера перед выполнением чтения, или просмотра поступающих сообщений с целью назначения им приоритетов для последующей обработки.

```
BOOL PeekNamedPipe (HANDLE hPipe, LPVOID lpBuffer, DWORD
cbBuffer, LPDWORD lpcbRead, LPDWORD lpcbAvail, LPDWORD
lpcbMessage);
```

Функция обеспечивает считывание любого байта или сообщения из канала без их удаления, но ее невозможно блокировать, и она осуществляет возврат сразу же по завершении выполнения.

Чтобы определить, имеются ли в канале данные, необходимо проверить значение ***lpcbAvail**; если данные в канале присутствуют, оно должно быть больше 0. В этом случае параметры **lpBuffer** и **cbRead** могут иметь значения NULL. Если же буфер определен параметрами **lpBuffer** и **cbBuffer**, то значение ***lpcbMessage** укажет, остается ли еще некоторое количество байтов сообщений, которые не умещаются в буфере. Для канала, работающего в режиме считывания байтов, это значение равно 0. Поскольку функция осуществляет чтение, не уничтожая данные, то для удаления данных из канала требуется последующее применение функции **ReadFile**.

В программе clientNT.c представлен однопоточной клиент, а в программе serverNT.c – сервер. Программы и файлы включения доступны на сайте в папке «WinAPI». Сервер соответствует модели, представленной на рис. 2.3. Запросом клиента является командная строка. Ответом сервера является результирующий вывод, который посылается в

виде нескольких сообщений. Система характеризуется следующими особенностями.

- С сервером могут взаимодействовать несколько клиентов.
- Клиенты могут находиться на различных системах в сети, хотя допускается и их расположение на компьютере сервера.
- Сервер является многопоточным, каждому именованному каналу назначается отдельный поток. Потоки предоставляются клиентам через экземпляр именованного канала, выделяемого клиенту.
- Отдельные потоки сервера в каждый момент времени обрабатывают один запрос.

2.4. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

На рис. 2.1 было показано, каким образом обеспечивается существование потоков в среде процесса. Использование потоков на примере многопоточного сервера, способного обрабатывать запросы одновременно нескольких клиентов, иллюстрирует рис. 2.3.

Потоки, принадлежащие одному процессу, разделяют общие данные и код, поэтому важно, чтобы каждый поток имел также собственную область памяти, относящуюся только к нему. В Windows удовлетворение этого требования обеспечивается несколькими способами.

- У каждого потока имеется собственный стек, который она использует при вызове функций и обработке некоторых данных.
- При создании потока вызывающий процесс может передать ему аргумент (**`Arg`** на рис. 2.4), который обычно является указателем. На практике этот аргумент помещается в стек потока.
- Каждый поток может распределять индексы собственных локальных областей хранения (TLS), считывать и устанавливать значения TLS. TLS предоставляют в распоряжение потоков небольшие массивы данных. TLS обеспечивают защиту данных одного потока от воздействия со стороны других потоков.

Для создания потоков в адресном пространстве процесса предусмотрен системный вызов **`CreateThread`**. Он требует указания:

- начального адреса потока в коде процесса;
- размера стека (из виртуального адресного пространства процесса), по умолчанию равен размеру стека основного потока;
- указателя на аргумент, передаваемый потоку.

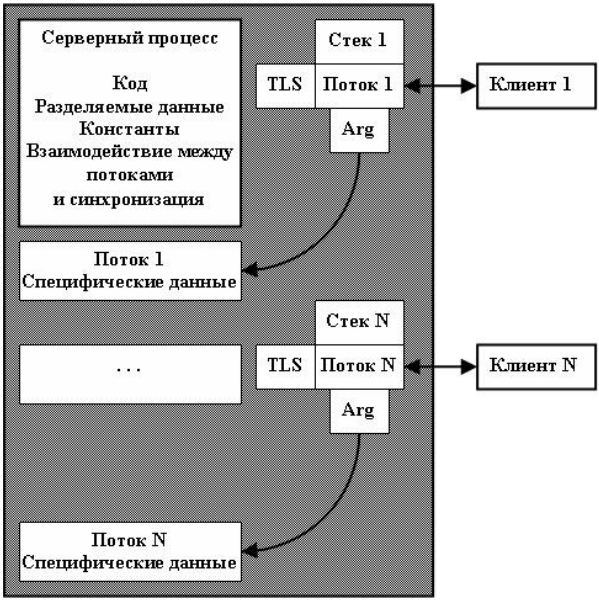

Рис. 2.4. Потоки в среде сервера

Функция возвращает значение идентификатора (ID) и дескриптор потока. В случае ошибки возвращаемое значение равно NULL.

```
HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpsa, DWORD
dwStackSize, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID
lpThreadParm,DWORD dwCreationFlags,LPDWORD lpThreadId);
```

Параметры

lpsa – указатель на структуру атрибутов защиты.

dwStackSize – размер стека нового потока в байтах. Значению 0 соответствует размер стека по умолчанию.

lpStartAddr – указатель на функцию (в коде процесса), которая должна выполняться. Функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Функция потока (**ThreadFunc**) имеет сигнатуру:

```
DWORD WINAPI ThreadFunc (LPVOID);
```

lpThreadParm – указатель, передаваемый в функцию потока.

dwCreationFlags – если значение этого параметра установлено равным 0, то поток запускается сразу после вызова функции **Create-**

Thread. Установка значения **CREATE_SUSPENDED** приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности вызовом функции **ResumeThread**.
lpThreadId – указатель на переменную типа DWORD, которая получает идентификатор нового потока.

```
VOID ExitThread (DWORD dwExitCode);
```

Поток процесса может завершить свое выполнение, вызвав функцию **ExitThread**, но обычным способом завершения потока является возврат из функции потока с использованием кода завершения в качестве возвращаемого значения. По завершении выполнения потока память, занимаемая его стеком, освобождается. Когда завершается выполнение последнего потока, завершается и выполнение процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции **TerminateThread**, однако освобождения ресурсов потока при этом не происходит.

Поток, выполнение которого было завершено, продолжает существовать до тех пор, пока посредством функции **CloseHandle** не будет закрыт его последний дескриптор. Любой другой поток может получить код завершения потока: **BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode)**. Если поток еще не завершен, то значение этой переменной будет равно **STILL_ACTIVE**.

Функции, используемые для получения идентификаторов и дескрипторов потоков, напоминают те, которые используются для аналогичных целей в случае процессов:

- **GetCurrentThread** – возвращает ненаследуемый псевдодескриптор вызывающего потока;
- **GetCurrentThreadId** – возвращает идентификатор потока;
- **GetThreadId** – возвращает идентификатор потока по дескриптору;
- **OpenThread** – создает дескриптор потока по идентификатору.

Для каждого потока поддерживается *счетчик приостановок* (suspend count), и выполнение потока может быть продолжено, если значение этого счетчика равно 0. Поток может увеличивать или уменьшать значение счетчика приостановок другого потока функциями **SuspendThread** и **ResumeThread**:

```
DWORD ResumeThread (HANDLE hThread);
DWORD SuspendThread (HANDLE hThread)
```

В случае успешного выполнения обе функции возвращают предыдущее значение счетчика приостановок, иначе – **0xFFFFFFFF**.

Поток может дожидаться завершения выполнения другого потока. При вызове функций ожидания (**WaitForSingleObject** и **WaitForMultipleObjects**) следует использовать дескрипторы потоков.

Допустимое количество объектов, одновременно ожидаемых функцией **WaitForMultipleObjects**, ограничено значением **MAXIMUM_WAIT_OBJECTS** (64), но при большом количестве потоков можно воспользоваться серией вызовов функций ожидания.

Функция ожидания дожидается, пока объект, указанный дескриптором, не перейдет в *сигнальное* состояние. Поток переводится в сигнальное состояние при помощи функций **ExitThread** и **TerminateThread**. Функция **ExitProcess** переводит в сигнальное состояние как сам процесс, так и все его потоки.

2.5. СРЕДСТВА СИНХРОНИЗАЦИИ ПОТОКОВ В WINDOWS

Windows предоставляет четыре объекта, предназначенных для синхронизации потоков и процессов. Три из них – мьютексы, семафоры и события – являются объектами ядра, имеющими дескрипторы. Четвертый объект – критические участки кода. Объекты критических участков кода являются предпочтительным механизмом, если их возможностей достаточно для удовлетворения требований программиста.

2.5.1. Критические участки кода

Объект критического участка кода – это участок программного кода, который каждый раз должен выполняться только одним потоком; параллельное выполнение этого участка несколькими потоками может приводить к непредсказуемым или неверным результатам.

Объекты **CRITICAL_SECTION** (CS) можно инициализировать и удалять, но они не имеют дескрипторов и не могут совместно использоваться другими процессами. Объекты должны объявляться как переменные типа **CRITICAL_SECTION**. Потоки входят в объекты CS и покидают их, но выполнение кода отдельного объекта CS каждый раз разрешено только одному потоку. Вместе с тем один и тот же поток может входить в несколько отдельных объектов CS и покидать их, если они расположены в разных местах программы.

Для инициализации и удаления переменной типа **CRITICAL_SECTION** используются, соответственно, функции:

```
VOID InitializeCriticalSection (
LPCRITICAL_SECTION lpCriticalSection)
VOID DeleteCriticalSection (
LPCRITICAL_SECTION lpCriticalSection)
```

Функция **EnterCriticalSection** блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того, как другой поток выполнит функцию **LeaveCriticalSection**. Говорят, что поток *получил права владения* объектом CS, если произошел возврат из функции **EnterCriticalSection**, тогда как для уступки прав владения используется функция **LeaveCriticalSection**.

```
VOID EnterCriticalSection (
LPCRITICAL_SECTION lpCriticalSection)
VOID LeaveCriticalSection (
LPCRITICAL_SECTION lpCriticalSection)
```

Поток, владеющий объектом CS, может повторно войти в этот же CS без его блокирования; таким образом, объекты CS являются *рекурсивными*. Поддерживается счетчик вхождений в объект CS, и поэтому поток должен покинуть CS столько раз, сколько было вхождений в него, чтобы разблокировать этот объект для других потоков. Выход из объекта CS, которым данный поток не владеет, может привести к непредсказуемым результатам, включая блокирование самого потока.

Для возврата из функции **EnterCriticalSection** нет конечного интервала ожидания; другие потоки будут блокированы на неопределенное время, пока поток, владеющий объектом CS, не покинет его. Используя функцию **TryEnterCriticalSection**, можно тестировать (опросить) CS, чтобы проверить, не владеет ли им другой поток:

```
BOOL TryEnterCriticalSection (LPCRITICAL_SECTION lpCri
ticalSection)
```

Возврат функцией **TryEnterCriticalSection** значения **True** означает, что вызывающий поток приобрел права владения критическим участком кода, тогда как возврат значения **False** говорит о том, что данный критический участок кода уже принадлежит другому потоку.

Объекты CS обладают тем преимуществом, что они не являются объектами ядра и поддерживаются в пользовательском пространстве.

Одним из наиболее распространенных способов применения объектов CS является обеспечение доступа потоков к разделяемым глобальным переменным. Рассмотрим пример:

```
CRITICAL_SECTION csl;
volatile DWORD N = 0, M;
/* N – глобальная переменная */
InitializeCriticalSection (&csl);
EnterCriticalSection (&csl);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&csl);
DeleteCriticalSection (&csl);
```

Полный пример применения критических секций реализован в программе simplePC.c, доступной на сайте в папке «WinAPI».

2.5.2. Мьютексы

Объект *взаимного исключения* (mutual exception), или *мьютекс* (mutex), обеспечивает более универсальную функциональность по сравнению с объектом **CRITICAL_SECTION**. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам.

Поток приобретает права владения мьютексом (*блокирует* мьютекс) путем вызова функции ожидания (**WaitForSingleObject** или **WaitForMultipleObjects**) по отношению к дескриптору мьютекса и уступает эти права посредством вызова функции **ReleaseMutex**.

Поток может завладевать одним и тем же ресурсом несколько раз, и при этом не будет блокироваться, если уже владеет данным ресурсом. Поток должен освободить мьютекс столько раз, сколько он его захватывал. Возможность рекурсивного захвата ресурсов полезна для ограничения доступа к рекурсивным функциям.

При работе с мьютексами используются функции **CreateMutex**, **ReleaseMutex** и **OpenMutex**.

```
HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpsa,
BOOL bInitialOwner, LPCTSTR lpMutexName)
```

bInitialOwner – если значение этого флага установлено равным **True**, вызывающий поток немедленно приобретает права владения новым мьютексом. Эта атомарная операция предотвращает приобретение прав владения мьютексом другими потоками, прежде чем это сделает поток, создающий мьютекс. Как следует из самого его названия (initial

owner – исходный владелец), этот флаг не оказывает никакого действия, если мьютекс уже существует.

lpMutexName – указатель на строку, содержащую имя мьютекса; в отличие от файлов имена мьютексов чувствительны к регистру. Если этот параметр равен NULL, то мьютекс создается без имени. События, мьютексы, семафоры и другие объекты ядра используют одно пространство имен, отличное от пространства имен файловой системы. Поэтому имена всех объектов синхронизации должны быть различными. Длина имен объектов не может превышать 260 символов.

Возвращаемое значение имеет тип **HANDLE**; значение NULL указывает на неудачное завершение функции.

```
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);
```

Функция **OpenMutex** открывает существующий именованный мьютекс, что дает возможность потокам, принадлежащим различным процессам, синхронизироваться, как если бы они принадлежали одному процессу. Вызову функции **OpenMutex** в одном процессе должен предшествовать вызов функции **CreateMutex** в другом процессе.

Функция **ReleaseMutex** освобождает мьютекс, которым владеет вызывающий поток. Если мьютекс не принадлежит потоку, функция завершается с ошибкой.

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Если поток, владевший мьютексом, завершился, не освободив его, то такой мьютекс называют *покинутым*. На то, что сигнализирующий дескриптор представляет покинутый мьютекс, указывает возврат функцией **WaitForSingleObject** значения **WAIT_ABANDONED_0** или использование значения **WAIT_ABANDONED_0** в качестве базового значения функцией **WaitForMultipleObject**.

2.5.3. Семафоры

Объекты второго из трех упомянутых типов объектов синхронизации ядра – *семафоры* поддерживают счетчики, и когда значение этого счетчика больше 0, объект семафора находится в сигнальном состоянии. Если же значение счетчика становится нулевым, объект семафора переходит в несигнальное состояние.

Потоки и процессы организуют ожидание, используя для этого одну или несколько функций. При разблокировании ожидающего потока значение счетчика уменьшается на 1.

К функциям управления семафорами относятся **CreateSemaphore**, **OpenSemaphore** и **ReleaseSemaphore**, причем последняя функция может инкрементировать значение счетчика на единицу и более. Эти функции аналогичны своим эквивалентам для управления мьютексами.

```
HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpsa,
LONG lSemInitial, LONG lSemMax, LPCTSTR lpSemName);
```

Параметр **lSemMax**, значение которого должно быть равным, по крайней мере, единице, определяет максимально допустимое значение счетчика семафора. Параметр **lSemInitial** – начальное значение этого счетчика, которое должно удовлетворять условию: 0 < **lSemInitial** < **lSemMax**. Возвращение функцией значения NULL указывает на ее неудачное выполнение.

Каждая отдельная операция ожидания может уменьшить значение счетчика только на единицу, но с помощью функции **ReleaseSemaphore** значение его счетчика может быть увеличено до любого значения вплоть до максимально допустимого.

```
BOOL ReleaseSemaphore (HANDLE hSemaphore, LONG cReleaseCount, LPLONG lpPreviousCount)
```

Есть возможность получения предыдущего значения счетчика, определяемого указателем **lpPreviousCount**, которое он имел до освобождения объекта синхронизации при помощи функции **ReleaseSemaphore**, но если необходимости в этом нет, то значение упомянутого указателя следует установить равным NULL.

Число, прибавляемое к счетчику семафора (**cReleaseCount**), должно быть больше 0, но если выполнение функции **ReleaseSemaphore** приводит к выходу значения счетчика за пределы допустимого диапазона, то она завершается с ошибкой, возвращая значение **FALSE**, а значение счетчика семафора остается неизменным. Предыдущим значением счетчика следует пользоваться с осторожностью, поскольку оно могло быть изменено другими потоками. Кроме того, невозможно определить, достиг ли счетчик максимально допустимого значения, поскольку не предусмотрено средство, отслеживающее увеличение счетчика в результате его освобождения.

Классической областью применения семафоров является управление распределением конечных ресурсов, когда значение счетчика семафора ассоциируется с определенным количеством доступных ресур-

сов, например, количеством сообщений, находящихся в очереди. Тогда максимальное значение счетчика соответствует максимальному размеру очереди. Таким образом, производитель помещает сообщение в буфер и вызывает функцию **ReleaseSemaphore**, обычно с увеличением значения счетчика на единицу (**cReleaseCount**). Потоки потребителя будут ожидать перехода семафора в сигнальное состояние, получая сообщения и уменьшая значения счетчика.

2.5.4. События

Последним из рассматриваемых типов объектов синхронизации ядра являются *события* (events). Объекты события используются для того, чтобы сигнализировать другим потокам о наступлении какого-либо события, например, о появлении нового сообщения.

Важной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции **CreateEvent**.

- Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.
- Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.

События используют пять новых функций: **CreateEvent**, **OpenEvent**, **SetEvent**, **ResetEvent** и **PulseEvent**.

```
HANDLE CreateEvent (LPSECURITY_ATTRIBUTES lpsa, BOOL
bManualReset, BOOL bInitialState, LPTCSTR lpEventName);
```

Чтобы создать событие, сбрасываемое вручную, необходимо установить значение параметра **bManualReset** равным **True**. Чтобы сделать начальное состояние события сигнальным, установите равным **True** значение параметра **bInitialState**. Для открытия именованного объекта события используется функция **OpenEvent**, причем это может сделать и другой процесс.

Для управления объектами событий используются три функции:

```
BOOL SetEvent (HANDLE hEvent);
BOOL ResetEvent (HANDLE hEvent);
BOOL PulseEvent (HANDLE hEvent);
```

Поток может установить событие в сигнальное состояние, используя функцию **SetEvent**. Если событие является автоматически сбрасываемым, то оно возвращается в несигнальное состояние после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии, пока такой поток не появится, после чего этот поток сразу же освобождается. Таким же образом ведет себя семафор, максимальное значение счетчика которого установлено равным единице.

Если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии, пока какой-либо поток не вызовет функцию **ResetEvent**, указав дескриптор этого события в качестве аргумента. Все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться.

Функция **PulseEvent** освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же использования автоматически сбрасываемого события функция **PulseEvent** освобождает только один ожидающий поток, если таковые имеются.

Функция **PulseEvent** становится полезной после того как сбрасываемое вручную событие установлено в сигнальное состояние с помощью функции **SetEvent**. При использовании функции **WaitForMultipleObjects** ожидающий поток освободится, когда одновременно *все* события будут находиться в сигнальном состоянии, но некоторые из событий, находящихся в сигнальном состоянии, могут быть сброшены, прежде чем поток освободится.

Комбинирование автоматически сбрасываемых и сбрасываемых вручную событий с функциями **SetEvent** и **PulseEvent** приводит к четырем различным способам использования событий. Каждая из четырех комбинаций, указанных в таблице 2.1, уникальна.

Таблица 2.1

Сводная таблица свойств событий

	Автоматически сбрасываемые события	**Сбрасываемые вручную события**
SetEvent	Освобождается строго один поток. Если ни один из потоков не ожидает наступления события, то поток, который первым перейдет в состояние ожидания следующих событий, будет освобожден. После этого событие автоматически сбрасывается	Освобождаются все потоки, которые в настоящее время ожидают наступления события. Событие остается в сигнальном состоянии до тех пор, пока не будет сброшено каким-либо потоком
PulseEvent	Освобождается строго один поток, но только в том случае, если имеется поток, ожидающий наступления события	Освобождаются все потоки, которые в этот момент ожидают наступления события, после чего событие сбрасывается и переходит в несигнальное состояние

Пример применения мьютексов и сигналов – программа eventPC.c, реализующая систему «производитель/потребитель» и доступная на сайте в разделе «WinAPI».

В качестве резюме перечислим в табл. 2.2 наиболее важные свойства объектов синхронизации Windows.

Таблица 2.2

Сравнительные характеристики объектов синхронизации

	CS	**Мьютекс**	**Семафор**	**Событие**
Именованный защищаемый объект	Нет	Да	Да	Да
Доступность из нескольких процессов	Нет	Да	Да	Да
Синхронизация	Вхождение	Ожидание	Ожидание	Ожидание

Окончание табл. 2.2

	CS	Мьютекс	Семафор	Событие
Освобождение	Выход	Освобождается или оставляется без контроля	Освобождается любым потоком	Функции SetEvent, PulseEvent
Права владения	Только один поток. Владелец может осуществлять вхождение несколько раз, не блокируя свое выполнение	Только один поток. Владелец может выполнять функцию ожидания несколько раз, не блокируя свое выполнение	Неприменимо. Доступ одновременно нескольким потокам, число ограничено значением счетчика	Неприменимо. Функции SetEvent и PulseEvent могут быть вызваны любым потоком
Результат освобождения	Разрешается вхождение одного потока из числа ожидающих	После освобождения права владения приобретает один поток из числа ожидающих	Продолжают выполнение несколько потоков, число определяется значением счетчика	Продолжает выполнение один или несколько ожидающих потоков

3. МНОГОЗАДАЧНОЕ И МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В Linux

3.1. ПРОЦЕССЫ В Linux

Процесс в ОС UNIX – это программа, выполняемая в собственном виртуальном адресном пространстве. Когда пользователь входит в систему, автоматически создается процесс, в котором выполняется программа командного интерпретатора. Если командному интерпретатору встречается команда, соответствующая выполняемому файлу, то он создает новый процесс и запускает в нем соответствующую программу. Эта запущенная программа в свою очередь может создать процесс и запустить в нем другую программу и т. д.

Для образования нового процесса и запуска в нем программы используются два системных вызова API – **`fork()`** и **`exec()`**. Системный вызов **`fork`** приводит к созданию нового адресного пространства, состояние которого абсолютно идентично состоянию адресного пространства основного процесса. Для дочернего процесса заводятся копии всех сегментов данных. После выполнения системного вызова **`fork`** управление и в основном, и в порожденном процессах находится в точке, непосредственно следующей за вызовом **`fork`**. Чтобы программа могла разобраться, в каком процессе она теперь работает – в основном или порожденном, функция **`fork`** возвращает 0 в порожденном процессе и целое положительное число (идентификатор порожденного процесса, PID) в основном процессе.

Для запуска новой программы в порожденном процессе нужно обратиться к системному вызову **`exec`**, указав в качестве аргументов имя файла, содержащего новую выполняемую программу, и текстовые строки – аргументы новой программы. Выполнение **`exec`** приводит к тому, что в адресное пространство порожденного процесса загружается новая выполняемая программа и запускается с адреса, соответствующего входу в функцию **`main`**.

Это приводит к замене текущего программного сегмента и текущего сегмента данных, которые были унаследованы при выполнении вызова **`fork`**, на новые соответствующие сегменты, заданные в файле. Прежние сегменты теряются. Это метод смены выполняемой процессом

программы, но не самого процесса. Файлы, уже открытые до выполнения **exec**, остаются открытыми после его выполнения.

Каждый процесс, за исключением начального, порождается в результате запуска другим процессом функции **fork**. Каждый процесс имеет одного родителя, но может породить много процессов. Начальный процесс создается в результате загрузки системы. После порождения процесса с идентификатором 1 (**init**) начальный процесс становится процессом подкачки и реализует механизм виртуальной памяти. Процесс **init** является предком любого другого процесса в системе и связан с каждым процессом.

Процесс может выполняться в одном из двух состояний: *пользовательском* и *системном.* В пользовательском состоянии процесс выполняет пользовательскую программу и имеет доступ к пользовательскому сегменту данных. В системном состоянии процесс выполняет программы ядра и имеет доступ к системному сегменту данных. Когда пользовательскому процессу требуется выполнить системную функцию, он создает системный вызов. Фактически происходит вызов ядра системы как подпрограммы.

С момента появления системного вызова процесс считается системным. Таким образом, пользовательский и системный процессы являются двумя фазами одного и того же процесса, но они никогда не пересекаются между собой. Каждая фаза пользуется своим собственным стеком. Стек задачи содержит аргументы, локальные переменные и другую информацию относительно функций, выполняемых в режиме задачи. Диспетчерский процесс не имеет пользовательской фазы.

В UNIX-системах используется разделение времени: каждому процессу выделяется квант времени. Процесс завершается сам до истечения отведенного ему кванта времени, либо он откладывается по истечении кванта. Все системные процессы имеют более высокие приоритеты по сравнению с пользовательскими и поэтому всегда обслуживаются в первую очередь.

3.2. МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ В LINUX

Стандартная библиотека C (libc, реализованная в Linux в glibc) использует возможности многозадачности Unix System V. Unix System V (далее SysV) – коммерческая реализация Unix, породившая одно из двух самых важных семейств Unix, второе – BSD Unix.

В libc тип **pid_t** определен как целое, способное вместить в себе PID. Будем использовать этот тип для работы с PID, хотя использование целого типа дало бы тот же результат. Рассмотрим функцию, которая сообщает PID процесса, содержащего нашу программу (она определена вместе с **pid_t** в файлах **unistd.h** и **sys/types.h**):

```
pid_t getpid (void);
```

Напишем программу, выводящую в стандартный вывод свой PID:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main() {
pid_t pid; pid = getpid();
printf("pid, присвоенный процессу – %d\n", pid);
return 0; }
```

Сохранив программу, скомпилируем ее:

```
gcc -Wall -o print_pid print_pid.c
```

Команда создаст исполняемый файл **print_pid**. Текущая директория не содержится в **path**, поэтому необходимо запустить программу как "**./print_pid**". Она выведет PID, и при последующих запусках это число будет постоянно увеличиваться, потому что в перерыве между запусками может быть создан другой процесс. Это можно выяснить, выполняя утилиту **ps** между запусками **print_pid**.

Скажем пару слов о ключах компилятора. Их очень много, в чем можно убедиться, выполнив **man gcc** или **info gcc**. Использованные ключи означают следующее: **-Wall**: выдавать все сообщения о предупреждениях и ошибках, **-o**: компилировать все перечисленные исходные файлы в указанный исполняемый файл.

3.2.1. Порождение процессов

Вот функция, которая создает новый процесс:

```
#include <unistd.h>
pid_t fork(void);
```

При ее вызове происходит разветвление выполнения процесса. Число, которое она возвращает – это pid. Текущий процесс дублируется в родительском и дочернем, которые будут выполняться, чередуясь с другими выполняющимися процессами. Решение, какой процесс должен выполняться, принимается диспетчером задач.

Оба процесса содержат коды, как родительские, так и дочерние, однако оба они должны выполнить только свой набор кодов. Чтобы прояснить это, взглянем на алгоритм:

```
РАЗВЕТВИТЬ
ЕСЛИ ТЫ ДОЧЕРНИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
ЕСЛИ ТЫ РОДИТЕЛЬСКИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
```

который представляет собой код, написанный на некоем метаязыке.

На языке С получим:

```
int main() {
  pid_t pid;
  pid = fork();
  if (pid == 0)
  { КОД ДОЧЕРНЕГО ПРОЦЕССА }
  КОД РОДИТЕЛЬСКОГО ПРОЦЕССА
}
```

Настоящий пример кода (fork_demo2.c) доступен на сайте [3] в папке «Linuxprog». Программа сделает разветвление и оба процесса, родительский и дочерний, выведут кое-что на экран. Вставляя задержку случайной длины перед каждым вызовом **prinf** при помощи функций **sleep** и **rand**, сможем нагляднее увидеть эффект многозадачности: `sleep(rand()%4);`

Это заставит программу «заснуть» на случайное число секунд: от 0 до 3 (% возвращает остаток от целочисленного деления).

3.2.2. Методы синхронизации процессов

Часто родительскому процессу необходимо синхронизироваться с дочерними, чтобы выполнять операции в нужное время. Основной способ синхронизации процессов – функции **wait** и **waitpid**.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

wait приостанавливает выполнение текущего процесса до завершения какого-либо из его процессов-потомков.

```
pid_t waitpid (pid_t pid, int *status, int options);
```

waitpid приостанавливает выполнение текущего процесса до завершения заданного процесса или проверяет завершение заданного процесса. Если процесс уже завершился, то приостанов текущего процесса не происходит. Здесь pid – это pid ожидаемого процесса, причем если pid > 0, то он задает pid процесса, завершение которого ожидает-

ся/проверяется функцией **waitpid**. Если pid = 0, то **waitpid** ожидает/проверяет завершение любого процесса той группы, к которой принадлежит текущий процесс. Если pid < 0, то **waitpid** ожидает/проверяет завершение любого процесса – своего потомка.

status – указатель на целое, которое будет содержать статус дочернего процесса (NULL, если эта информация не нужна), а **options** – это набор опций, задающих режим поведения **waitpid**. Может задаваться одним из следующих значений или их логическим ИЛИ:

- **WNOHANG** – не приостанавливать текущий процесс, если проверяемый процесс не завершился;
- **WUNTRACED** – не приостанавливать текущий процесс для потомков, которые завершились, но о состоянии которых еще не доложено.

Значение **options** = 0 определяет переход в ожидание, если проверяемый процесс не завершился.

Если **status** не равен NULL, то функции **wait** и **waitpid** сохраняют информацию о статусе в переменной, на которую указывает **status**. Этот статус можно проверить с помощью нижеследующих макросов (они принимают в качестве аргумента буфер (типа int)).

- **WIFEXITED (status)** не равно нулю, если дочерний процесс успешно завершился.
- **WIFEXITEDSTATUS (status)** возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main(). Этот макрос можно использовать, только если **WIFEXITED** вернул ненулевое значение.
- **WIFSIGNALED (status)** возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.
- **WTERMSIG (status)** возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если **WIFSIGNALED** вернул ненулевое значение.
- **WIFSTOPPED (status)** возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг **WUNTRACED**.
- **WSTOPSIG (status)** возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если **WIFSTOPPED** вернул ненулевое значение.

Пример программы (fork_demo3.c), где родительский процесс создает дочерний и ждет его завершения, доступен на сайте в той же папке.

3.2.3. Функции и программы в порожденных процессах

Чтобы в качестве дочернего процесса вызвать функцию, достаточно вызвать ее после ветвления:

```
pid = fork();
if (pid == 0) {
// если дочерний процесс, то вызовем функцию
pid=process(arg);
// выход из дочернего процесса
exit(0);
}
```

Часто в качестве дочернего процесса необходимо запускать другую программу. Для этого применяется функции семейства exec:

```
#include <unistd.h>
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ...,
char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

Первый аргумент всех функций является указателем на символьную строку, содержащую полное имя исполняемого файла (path). Для функций **execlp** и **execvp** имя файла может задаваться без пути (file). Если первый аргумент этих функций не содержит символов."/", то файл ищется в каталогах, определенных в переменной окружения PATH.

Аргументы **arg**,... функций **execl**, **exelp**, **execle** составляют список указателей на символьные строки, содержащие параметры, передаваемые программе. По соглашениям первый элемент этого списка должен содержать имя программного файла. Список параметров должен заканчиваться пустым указателем – NULL или **(char *)0**.

В функциях **execv** и **execvp** параметры, передаваемые программе, передаются через массив символьных строк. Аргумент **argv** является указателем на этот массив.

Аргумент **envp** функции **execle** также является массивом указателей на символьные строки. Эти строки представляют собой окружение – среду для нового образа процесса. Последний элемент массива **envp** должен быть пустым указателем.

При нормальном выполнении функции возвращают 0. При ошибках выполнения функции возвращают –1 и устанавливают значение переменной **errno**, по которому можно определить причину ошибки.

Пример – программа spaces.c на сайте в папке «API ОС».

Если необходимо узнать состояние порожденного процесса при его завершении и возвращенное им значение, то используют макрос **WEXITEDSTATUS**, передавая ему в качестве параметра статус дочернего процесса.

```
stat=waitpid(pid,&status,WNOHANG);
if (pid == stat) {
printf("PID: %d, Result = %d\n", pid, WEXITSTATUS
(status)); }
```

3.2.4. Управление приоритетами процессов

Для управления приоритетами процессов применяются функции **getpriority** и **setpriority**:

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

getpriority читает приоритет процесса, группы процессов или пользователя и возвращает текущий приоритет заданного объекта. **setpriority** устанавливает приоритет процесса, группы процессов или пользователя. Параметр **which** определяет, должен ли системный вызов работать с приоритетом процесса, группы процессов или пользователя, его возможные значения – PRIO_PROCESS, PRIO_PGRP или PRIO_USER, соответственно. Параметр **who** задает идентификатор процесса, группы процессов или пользователя – в зависимости от значения параметра **which**. Параметр **prio** задает значение приоритета – число в пределах от –20 до +20. Большее приоритетное число означает низший реальный приоритет процесса. Начальное значение приоритета – 0. Только суперпользователь (root) может устанавливать для своего процесса приоритет выше текущего.

Функция **setpriority** при успешном завершении возвращает 0, иначе – возвращает –1, и устанавливает код ошибки в **errno**.

Пример управления приоритетами – программа prior.c в папке «Linuxprog» на сайте.

3.2.5. Уничтожение процессов

Для уничтожения процесса служит функция **kill**:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Если pid > 0, то он задает pid процесса, которому посылается сигнал. Если pid = 0, то сигнал посылается всем процессам той группы, к которой принадлежит текущий процесс. **sig** – тип сигнала.

Рассмотрим некоторые типы сигналов в Linux:

- **SIGKILL –** сигнал немедленного завершения процесса. Этот сигнал процесс не может игнорировать;
- **SIGTERM** сигнал запроса на завершение процесса;
- **SIGCHLD** сигнал системы процессу после запрошенного завершения одного из его дочерних процессов.

Раскомментировав вызов функции **kill** в предыдущем примере и запустив эту программу несколько раз, увидим, что процессы с большим временем спячки и низким приоритетом не успевают проработать до нормального завершения.

3.3. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ИНФОРМАЦИИ ПРОЦЕССАМИ

В модели программирования Unix в системе могут одновременно выполняться несколько процессов, каждому из которых выделяется собственное адресное пространство. Это иллюстрирует рис. 3.1.

Два процесса в левой части совместно используют информацию, хранящуюся в одном из объектов файловой системы. Для доступа к этим данным каждый процесс должен обратиться к ядру (используя функции **read**, **write**, **seek** и аналогичные).

Два процесса в середине рисунка совместно используют информацию, хранящуюся в ядре. Примерами являются канал, очередь сообщений или семафор SysV. Для доступа к совместно используемой информации в этом случае будут использоваться системные вызовы.

Два процесса в правой части используют общую область памяти, к которой может обращаться каждый из процессов. После того как будет

получен доступ к этой области памяти, процессы смогут обращаться к данным вообще без помощи ядра. Процессам, использующим общую память, также требуется синхронизация.

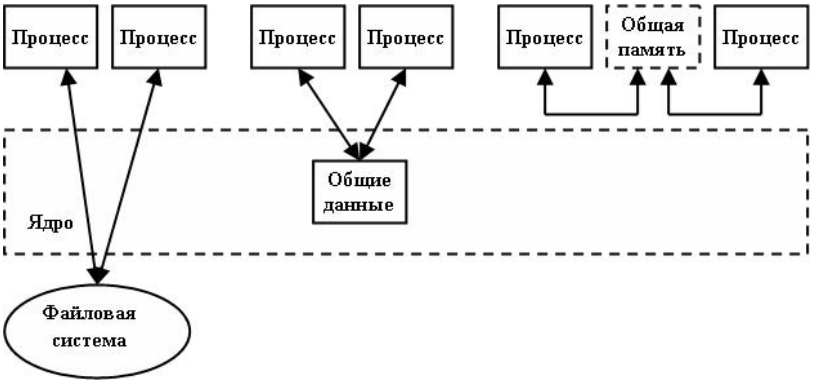

Рис. 3.1. Совместное использование информации процессами

Такое взаимодействие процессов часто называют аббревиатурой IPC (InterProcess Communication, межпроцессное взаимодействие) [5].

Исторически первым средством взаимодействия процессов в ОС UNIX являлись неименованные (pipes) и именованные (FIFO) программные каналы. Именованный канал служит для общения и синхронизации произвольных процессов, знающих имя данного канала и имеющих соответствующие права доступа. Неименованным каналом могут пользоваться только создавший его процесс и его потомки.

Существует два подхода к организации IPC: SysV IPC и POSIX IPC.

Первый является «родным» для UNIX и с его помощью реализовано большое количество существующих приложений. Второй призван обеспечить переносимость программного обеспечения.

В SysV IPC входят:

- очереди сообщений System V;
- семафоры System V;
- общая память System V.

У них много общего: схожи функции, с помощью которых организуется доступ к объектам; также схожи формы хранения информации в ядре. Информация о функциях сведена в табл. 3.1.

Таблица 3.1

Функции System V IPC

	Очереди сообщений	Семафоры	Общая память
Заголовочный файл	<sys/ msg.h>	<sys/ sem.h>	<sys/ shm.h>
Создание или открытие	msgget	semget	shmget
Операции управления	msgctl	semctl	shmctl
Операции IPC	msgsnd, msgrcv	semop	shmat, shmdt

Из имеющихся типов IPC следующие три могут быть отнесены к POSIX IPC:

- очереди сообщений POSIX IPC;
- семафоры POSIX IPC;
- общая память POSIX IPC.

Информация о функциях сведена в табл. 3.2.

Таблица 3.2

Функции POSIX IPC

	Очереди сообщений	Семафоры	Общая память
Заголовочный файл	<mqueue. h>	<semaphore.h>	<sys/ mman.h>
Создание или открытие	mq_open	sem_open	shm_open
Операции управления	mq_getattr, mq_setattr	–	fstat
Операции IPC	mq_send, mq_receive	sem_wait, sem_post	mmap, munmap

Основное отличие этих подходов заключается в способах создания идентификаторов:

- в System V используются ключи типа **`key_t`** и функция **`ftok`**;
- в POSIX используются имена, аналогичные именам файлов в файловой системе, но не обязанные соответствовать реальным файлам.

Далее рассматривается применение неименованных и именованных каналов, а также объектов SYS V IPC.

3.2.1. Неименованные каналы

Неименованные каналы – это самая первая форма IPC в Unix, появившаяся еще в 1973 году. Главным недостатком неименованных каналов является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. Это было исправлено в Unix System III (1982) добавлением каналов FIFO, которые еще называются именованными каналами.

Проиллюстрируем использование программных каналов, FIFO и очередей сообщений SysV приложением модели «клиент-сервер» со структурой, приведенной на рис. 3.2. Клиент считывает полное имя (файла) из стандартного потока ввода и записывает его в канал IPC. Сервер считывает это имя из канала IPC и производит попытку открытия файла на чтение. Если попытка оказывается успешной, сервер считывает файл и записывает его в канал IPC. В противном случае сервер возвращает клиенту сообщение об ошибке. Клиент считывает данные из канала IPC и записывает их в стандартный поток вывода. Если сервер не может считать файл, из канала будет считано сообщение об ошибке. В противном случае будет принято содержимое файла. Две штриховые линии на рис. 3.2 представляют собой канал IPC.

Рис. 3.2. Структура приложения модели «клиент-сервер»

Неименованные каналы имеются во всех существующих реализациях и версиях Unix. Канал создается вызовом pipe и предоставляет возможность однонаправленной (односторонней) передачи данных:

```
#include <unistd.h>
int pipe(int fd[2]);
```

и возвращает 0 в случае успешного завершения, –1 – в случае ошибки. Функция возвращает два файловых дескриптора: fd[0] и fd[1], причем первый открыт для чтения, а второй – для записи.

На рис. 3.3 изображен канал при использовании его единственным процессом.

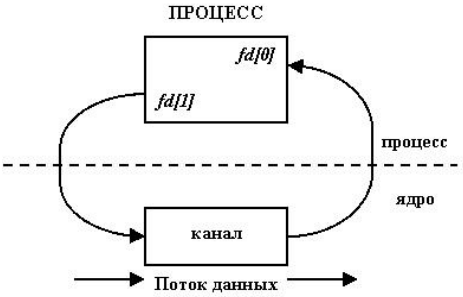

Рис. 3.3. Канал при использовании его единственным процессом

Каналы обычно используются для связи между двумя процессами (родительским и дочерним): процесс создает канал, а затем вызывает **`fork`**, создавая свою копию – дочерний процесс (рис. 3.4). Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний в свою очередь – открытый на запись конец канала.

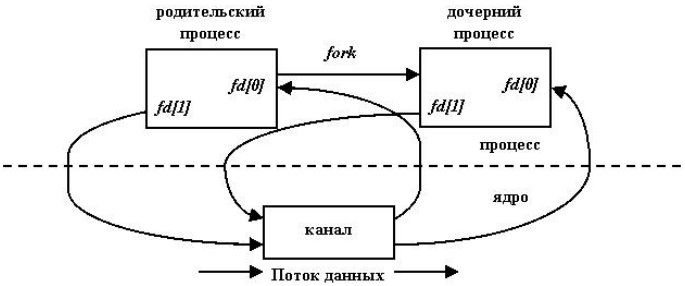

Рис. 3.4. Канал при использовании его родственными процессами

Это обеспечивает одностороннюю передачу данных между процессами, как показано на рис. 3.5.

При вводе команды наподобие **`who | sort | lp`** в интерпретаторе команд Unix интерпретатор выполняет действия для создания трех

процессов с двумя каналами между ними. Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись – к стандартному потоку вывода. Созданный при этом канал (конвейер) изображен на рис. 3.6.

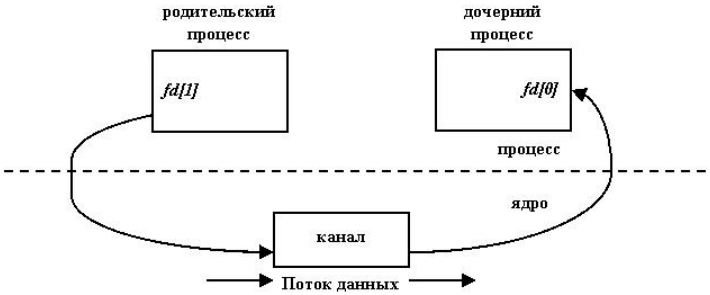

Рис. 3.5. Односторонняя передача данных через канал

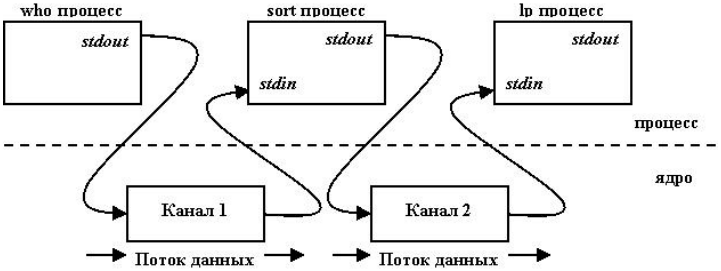

Рис. 3.6. Конвейерная передача данных через канал

Все перечисленные каналы были однонаправленными, то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного канала IPC следующие.

- Создаются каналы 1 (fdl[0] и fdl[l]) и 2 (fd2[0] и fd2[l]).
- Вызов **`fork`**.
- Родительский процесс закрывает доступный для чтения конец канала 1 (fdl [0]) и доступный для записи конец канала 2 (fd2[l]).
- Дочерний процесс закрывает доступный для записи конец канала 1 (fd1[1]) и доступный для чтения конец канала 2 (fd2[0]).

Текст программы mainpipe.c доступен на сайте в папке «Pipes». При этом создается структура каналов, изображенная на рис. 3.7.

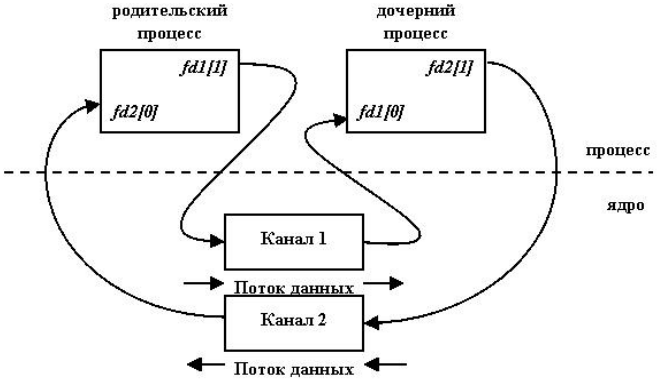

Рис. 3.7. Двунаправленная передача данных через каналы

Ниже приведен результат работы программы в случае наличия файла с указанным полным именем и в случае возникновения ошибок:

```
[gun@gun_linux_vm pipes]$ ./mainpipe
/etc/yum.conf    файл из нескольких строк
[main]
cachedir=/var/cache/yum
mirrors=/var/cache/yum/mirrors/
[gun@gun_linux_vm pipes]$ ./mainpipe
/etc/shadow      файл, на чтение которого нет прав
/etc/shadow: can't open  Permission denied
[gun@gun_linux_vm pipes]$ ./mainpipe
/no/such/file    несуществующий файл
/no/such/file: can't open  No such file or directory
```

Другим примером использования каналов является имеющаяся в стандартной библиотеке ввода-вывода функция **popen**, которая создает канал и запускает другой процесс, записывающий данные в этот канал или считывающий их из него:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Аргумент *command* представляет собой команду интерпретатора. Он обрабатывается программой sh (интерпретатор Bourne shell), поэтому для поиска исполняемого файла, вызываемого командой *command,*

используется переменная PATH. Канал создается между вызывающим процессом и указанной командой. Возвращаемое функцией **popen** значение представляет собой NULL – в случае ошибки или обычный указатель на тип FILE, который может использоваться для ввода или для вывода в зависимости от содержимого строки *type:*

- если *type* имеет значение **r**, вызывающий процесс считывает данные, направляемые командой *command* в стандартный поток вывода;
- если *type* имеет значение **w**, вызывающий процесс записывает данные в стандартный поток ввода команды *command.*

Функция **pclose** закрывает стандартный поток ввода-вывода *stream,* созданный командой **popen**, ждет завершения работы программы и возвращает код завершения, принимаемый от интерпретатора или –1 – в случае ошибки.

В тексте программы mainpopen.c, доступной на сайте в той же папке, дано еще одно решение задачи с клиентом и сервером, использующее функцию **popen** и программу (утилиту Unix) **cat**.

Одним из отличий этой реализации от mainpipe.c является отсутствие возможности формировать собственные сообщения об ошибках. Теперь мы целиком зависим от программы cat, а выводимые ею сообщения не всегда адекватны. Программа **cat** записывает сообщение об ошибке в стандартный поток сообщений об ошибках (stderr), а **popen** с этим потоком не связывается – к создаваемому каналу подключается только стандартный поток вывода.

3.2.2. Именованные каналы

Главным недостатком неименованных каналов является невозможность передачи информации между неродственными процессами. Два неродственных процесса не могут создать канал для связи между собой (если не передавать дескриптор).

Аббревиатура FIFO расшифровывается как «first in, first out» – «первым вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в Unix функционируют подобно неименованным – они позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO.

FIFO создается функцией mkfifo:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Она возвращает 0 при успешном выполнении –1 – при возникновении ошибок. Здесь *pathname* – полное имя файла, которое и будет именем FIFO. Аргумент *mode* указывает битовую маску разрешений доступа к файлу, аналогично второму аргументу команды open. В заголовке **<sys/stat.h>** определены шесть констант, которые могут использоваться для задания разрешений доступа к FIFO.

Функция **mkfifo** действует как **open**, вызванная с аргументом **O_CREAT | O_EXCL**. Это означает, что создается новый канал FIFO или возвращается ошибка **EEXIST**, в случае если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, вызывайте open вместо **mkfifo**. Для открытия существующего канала или создания нового в том случае, если его еще не существует, вызовите **mkfifo**, проверьте, не возвращена ли ошибка **EEXIST**, и если такое случится, вызовите функцию **open**.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции **open**, либо **fopen,** но не на чтение и запись, поскольку каналы FIFO могут быть только односторонними.

При записи в программный канал или канал FIFO вызовом **write** данные всегда добавляются к уже имеющимся, а вызов **read** считывает данные, помещенные в программный канал или FIFO первыми. При вызове функции **lseek** для программного канала или FIFO будет возвращена ошибка **ESPIPE**.

Переделаем программу mainpipe.c, чтобы использовать каналы FIFO вместо двух программных каналов. Функции client и server останутся прежними; отличия появятся только в функции main, новый текст которой приведен в файле mainfifo.c, доступном в той же папке сайта.

Константа FILE_MODE определена в нем как

```
#define FILE_MODE(S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
```

Владельцу файла разрешается чтение и запись в него, а группе и прочим пользователям – только чтение. Эти биты разрешений накладываются на маску режима доступа создаваемых файлов процесса.

Изменения по сравнению с примером, в которым использовались программные каналы, следующие:

- Для создания и открытия программного канала требуется только один вызов – **pipe**. Для создания и открытия FIFO требуется вызов **mkfifo** и последующий вызов **open**.
- Программный канал автоматически исчезает после того как будет закрыт последним использующим его процессом. Канал FIFO удаляется из файловой системы только после вызова **unlink**.

Картина аналогична примеру с каналами и иллюстрируется рис. 3.8.

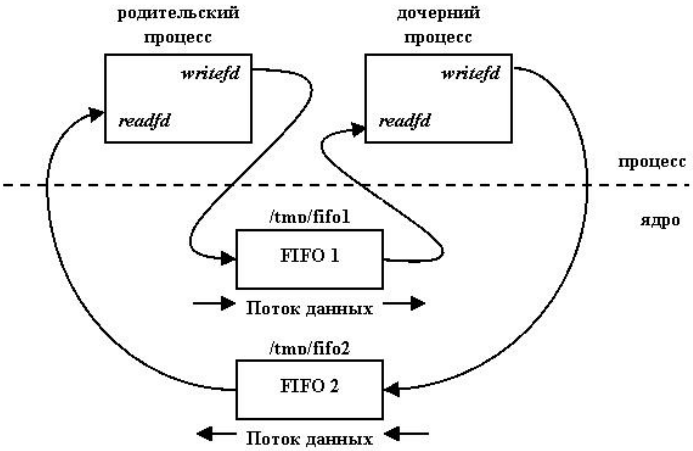

Рис. 3.8. Двунаправленная передача данных через каналы FIFO

Польза от лишнего вызова, необходимого для создания FIFO, следующая: канал FIFO получает имя в файловой системе, что позволяет одному процессу создать такой канал, а другому открыть его, даже если последний не является родственным первому.

В программах, некорректно использующих каналы FIFO, могут возникать неочевидные проблемы. Если поменять порядок двух вызовов функции open в породившем процессе в файле mainfifo.c, программа перестанет работать. Причина в том, что чтение из FIFO блокирует процесс, если канал еще не открыт на запись другим процессом.

В mainfifo.c клиент и сервер все еще являлись родственными процессами. Переделаем этот пример так, чтобы родство между ними отсутствовало. В файле server_main.c приведен текст программы-сервера.

Текст идентичен той части программы из mainfifo.c, которая относилась к серверу. Заголовочный файл fifo.h определяет имена двух FIFO, которые должны быть известны как клиенту, так и серверу. В файле client_main.c приведен текст программы-клиента, которая мало отличается от части программы из mainfifo.c, относящейся к клиенту. Заметим, что именно клиент, а не сервер удаляет канал FIFO по завершении работы, потому что последние операции с этим каналом выполняются им. Все файлы доступны в той же папке на сайте.

Некоторые свойства именованных и неименованных каналов заслуживают более пристального внимания. Прежде всего можно сделать дескриптор канала неблокируемым двумя способами.

При вызове **open** указать флаг **O_NONBLOCK**. Например, первый вызов **open** в client_main.c мог бы выглядеть так:

```
writefd = Open(FIFO1, O_WRONLY | O_NONBLOCK, 0);
```

Если дескриптор уже открыт, можно использовать **fcntl** для включения флага **O_NONBLOCK**. Эта функция применима для программных каналов, поскольку для них не вызывается функция **open** и нет возможности указать флаг **O_NONBLOCK** при ее вызове. Используя функцию **fcntl**, получим текущий статус файла с помощью **F_GETFL**, затем добавлим к нему с помощью побитового логического сложения (OR) флаг **O_NONBLOCK** и запишем новый статус командой **F_SETFL**:

```
int flags;
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
err_sys("F_GETFL error");
flags != O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
err_sys("F_SETFL error");
```

Будьте аккуратны с программами, которые просто устанавливают требуемый флаг, поскольку при этом сбрасываются все прочие флаги:

```
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
err_sys ("F_SETFL error");
```

Таблица 3.3 иллюстрирует действие флага, отключающего блокировку, при открытии и при чтении данных из пустого программного канала или канала FIFO.

Упомянем несколько дополнительных правил, действующих при чтении и записи данных в программные каналы и FIFO.

- Попытка считать больше данных, чем в данный момент содержится в канале, возвращает только имеющийся объем данных. Нужно предусмотреть обработку ситуации, в которой функция **read** возвращает меньше данных, чем было запрошено.

Таблица 3.3

Действие флага `O_NONBLOCK`

Операция	**Наличие открытых каналов**	**Блокировка включена (по умолчанию)**	**Флаг `O_NONBLOCK` установлен**
Открытие FIFO только для чтения	FIFO открыт на запись	Возвращается код успешного завершения операции	Возвращается код успешного завершения операции
Открытие FIFO только для чтения	FIFO не открыт на запись	Блокируется, пока FIFO не будет открыт на запись	Возвращается код успешного завершения операции
Открытие FIFO только для записи	FIFO открыт на чтение	Возвращает код успешного завершения операции	Возвращает код успешного завершения операции
Открытие FIFO только для записи	FIFO не открыт на чтение	Блокируется до тех пор, пока FIFO не будет открыт на чтение	Возвращает ошибку с кодом ENXIO
Чтение из пустого программного канала или FIFO	Программный канал или FIFO открыт на запись	Блокируется, пока в канал не будут помещены данные или канал не будет закрыт всеми процессами, которыми он был открыт на запись	Возвращает ошибку с кодом EAGAIN
Чтение из пустого программного канала или FIFO	Канал не открыт на запись	read возвращает 0 (конец файла)	read возвращает 0 (конец файла)
Запись в программный канал или FIFO	Канал открыт на чтение	(См. в тексте)	(См. в тексте)
Запись в программный канал или FIFO	Канал не открыт на чтение	Программе посылается сигнал SIGPIPE	Программе посылается сигнал SIGPIPE

- Если количество байтов, направленных на запись функции **write**, не превышает значения **PIPE_BUF**, то ядро гарантирует атомарность операции записи. Это означает, что если два процесса запишут данные в канал приблизительно одновременно, то в буфер будут помещены сначала все данные от первого процесса, а затем от второго, либо наоборот. Данные от двух процессов при этом не будут смешиваться. Однако если количество байтов превышает значение **PIPE_BUF**, атомарность операции записи не гарантируется.

Установка флага **O_NONBLOCK** не влияет на атомарность (способность выполняться, как единое целое) операции записи в канал – она определяется объемом посылаемых данных в сравнении с величиной **PIPE_BUF**. Однако если для канала отключена блокировка, возвращаемое функцией **write** значение зависит от количества байтов, отправленных на запись, и наличия свободного места в канале. Если количество байтов не превышает величины **PIPE_BUF**, то:

- если в канале достаточно места для записи требуемого количества данных, они будут переданы все сразу;
- если места в программном канале или FIFO недостаточно для записи требуемого объема данных, происходит немедленное завершение работы функции с возвратом ошибки **EAGAIN**.

Если количество байтов превышает значение **PIPE_BUF**, то:

- если в программном канале или FIFO есть место хотя бы для одного байта, ядро передает в буфер столько данных, сколько туда может поместиться, и это количество возвращается функцией **write**;
- если в программном канале или FIFO свободное место отсутствует, происходит завершение работы с возвратом ошибки **EAGAIN**.

При записи в программный канал или FIFO, не открытый для чтения, ядро посылает сигнал **SIGPIPE**:

- если процесс не принимает (**catch**) и не игнорирует **SIGPIPE**, выполняется действие по умолчанию – завершение работы процесса;
- если процесс игнорирует сигнал **SIGPIPE** или перехватывает его и возвращается из подпрограммы его обработки, **write** возвращает ошибку с кодом **EPIPE**.

3.2.3. Введение в System V IPC

Заголовочный файл **<sys/types.h>** определяет тип **key_t** как целое (по меньшей мере, 32-разрядное). Значения переменным этого типа обычно присваиваются функцией **ftok**. Функция **ftok** преобразовывает существующее полное имя и целочисленный идентификатор в значение типа **key_t** (называемое ключом IPC – IPC key).

```
#include <sys/ipc.h>
key_t ftok(const char *name, int id);
```

Функция использует полное имя файла и младшие 8 бит идентификатора для формирования целочисленного ключа IPC. Функция возвращает ключ IPC либо –1 при возникновении ошибки. Функция предполагает, что для конкретного приложения, использующего IPC, клиент и сервер используют одно и то же полное имя файла. Если клиенту и серверу для связи требуется только один канал IPC, идентификатору можно присвоить, например, значение 1. Если требуется несколько каналов IPC (например, один от сервера к клиенту и один в обратную сторону), идентификаторы должны иметь разные значения: например, 1 и 2. После того как клиент и сервер договорятся о полном имени и идентификаторе, они оба вызывают функцию **ftok** для получения одинакового ключа IPC.

Большинство реализаций функции **ftok** вызывают функцию **stat**, а затем объединяют:

- информацию о файловой системе, к которой относится полное имя pathname (поле **st_dev** структуры **stat**);
- номер узла (i-node) в файловой системе (поле **st_ino** структуры **stat**);
- младшие 8 битов идентификатора (не должен равняться нулю!).

Из комбинации этих трех значений получается 32-разрядный ключ. Номер узла (i-node) всегда отличен от нуля, поэтому большинство реализаций определяют константу **IPC_PRIVATE** равной нулю. Если указанное имя файла не существует или недоступно вызывающему процессу, **ftok** возвращает значение –1. Файл, имя которого используется для вычисления ключа, не должен быть одним из тех, которые создаются и удаляются в процессе работы, поскольк-

ку каждый раз при создании заново эти файлы получают другой номер узла, а это может изменить ключ, возвращаемый функцией **ftok** при очередном вызове.

Программа ftok.c, доступная для скачивания на сайте в папке «System V IPC», принимает полное имя в качестве аргумента командной строки, вызывает функции **stat** и **ftok**, затем выводит значения полей **st_dev** и **st_ino** структуры **stat** и получающийся ключ IPC. Эти три значения выводятся в шестнадцатеричном формате, поэтому легко видеть, как именно ключ IPC формируется из этих двух значений и идентификатора 0x57.

Для каждого объекта IPC, как для обычного файла, в ядре хранится набор информации, объединенной в структуру.

```
struct ipc_perm {
uid_t uid; /id пользователя владельца*/
gid_t gid; /id группы владельца */
uid_t cuid; /id создателя*/
gid_t cgid; /id группы создателя*/
mode_t mode; /*разрешения чтения-записи*/
ulong_t seq; /Последовательный номер канала*/
key_t key; /* ключ IPC */ };
```

Эта структура вместе с другими поименованными константами для функций System V IPC определена в файле **<sys/ipc.h>**.

Три функции **getXXX**, используемые для создания или открытия объектов IPC (табл. 3.1), принимают ключ IPC (типа **key_t**) в качестве одного из аргументов и возвращают целочисленный идентификатор. У приложения есть две возможности задания ключа (первого аргумента функций **getXXX**):

- вызвать **ftok**, передать ей полное имя и идентификатор;
- указать в качестве ключа константу **IPC_PRIVATE**, гарантирующую создание нового уникального объекта IPC.

Последовательность действий иллюстрирует рис. 3.9.

Все три функции **getXXX** (табл. 3.1) принимают в качестве второго аргумента набор флагов **oflag**, задающий биты разрешений чтения-записи (поле **mode** структуры **ipc_perm**) для объекта IPC и определяющий, создается ли новый объект IPC или производится обращение к уже существующему. Для этого имеются следующие правила.

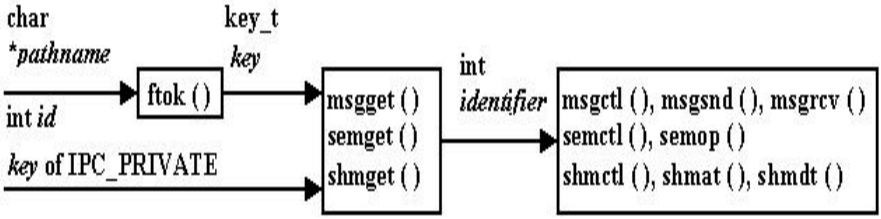

Рис. 3.9. Создание и открытие каналов IPC

- Ключ **IPC_PRIVATE** гарантирует создание уникального объекта IPC. Никакие возможные комбинации полного имени и идентификатора не могут привести к тому, что функция **ftok** вернет в качестве ключа значение **IPC_PRIVATE**.
- Установка бита **IPC_CREAT** аргумента **oflag** приводит к созданию новой записи для указанного ключа, если она еще не существует. Если запись существует, то возвращается ее идентификатор.
- Одновременная установка битов **IPC_CREAT** и **IPC_EXCL** аргумента **oflag** приводит к созданию новой записи для указанного ключа только в том случае, если такая запись еще не существует. Если же запись существует, то функция возвращает ошибку **EEXIST**.
- Комбинация **IPC_CREAT** и **IPC_EXCL** для объектов IPC действует аналогично комбинации **0_CREAT** и **0_EXCL** для функции **open**.
- Установка бита **IPC_EXCL** без **IPC_CREAT** никакого эффекта не дает.

Логическая диаграмма последовательности действий при открытии объекта IPC изображена на рис. 3.10. В табл. 3.4 показан альтернативный взгляд на этот процесс.

Обратите внимание, что в средней строке табл. 3.4 для флага **IPC_CREAT** без **IPC_EXCL** мы не получаем никакой информации о том, был ли создан новый объект или получен доступ к существующему. Для большинства приложений характерно создание сервером объекта IPC с указанием **IPC_CREAT** или **IPC_CREAT | IPC_EXCL**.

При создании нового объекта IPC с помощью одной из функций **getXXX**, вызванной с флагом **IPC_CREAT**, в структуру **ipc_perm** заносится следующая информация.

1. Часть битов аргумента **oflag** задают значение поля **mode** структуры **ipc_perm**. В табл. 3.5 указаны биты разрешений для трех типов IPC.

2. Поля **cuid** и **cgid** получают значения, равные действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами создателя.

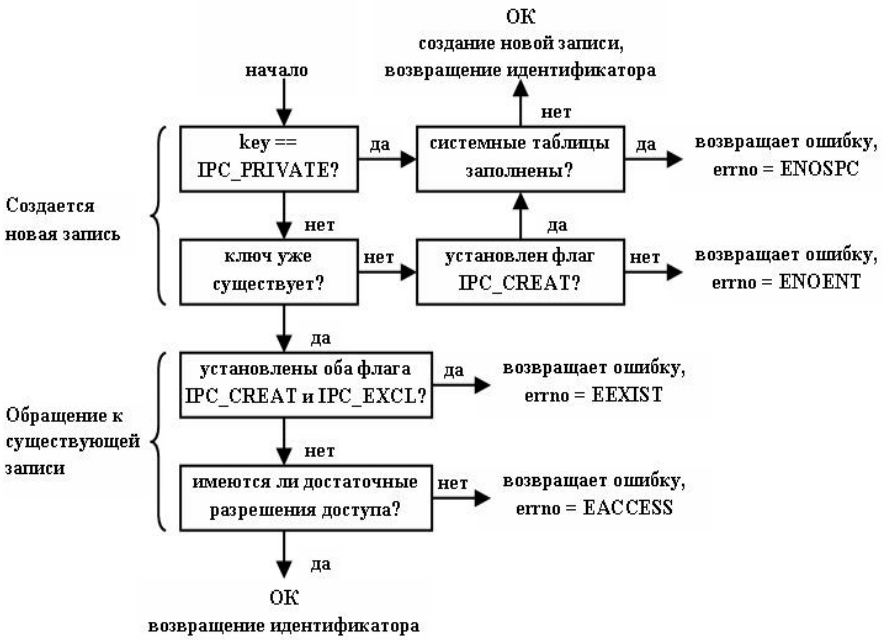

Рис. 3.10. Последовательность действий при открытии объекта IPC

Таблица 3.4

Последовательность действий при открытии объекта IPC

Аргумент **oflag**	Ключ не существует	Ключ существует
Специальные флаги не установлены	Ошибка, **errno** = **ENOENT**	ОК, открытие существующего объекта
IPC_CREAT	ОК, создается новая запись	ОК, открытие существующего объекта
IPC_CREAT|IPC_EXCL	ОК, создается новая запись	Ошибка, **errno** = **EEXIST**

3. Поля **uid** и **gid** устанавливаются равными действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами владельца.

Идентификатор создателя изменяться не может, тогда как идентификатор владельца может быть изменен процессом с помощью вызова функции **ctlXXX** для данного типа IPC с командой **IPC_SET**. Три функции **ctlXXX** позволяют процессу изменять биты разрешений доступа (поле **mode**) объекта IPC.

Таблица 3.5

Биты разрешений для трех типов IPC

Число (octal)	Очередь сообщений	Семафор	Разделяемая память	Описание
0400	MSG_R	SEM_R	SHM_R	Пользователь – чтение
0200	MSG_W	SEM_A	SHM_W	Пользователь – запись
0040	MSG_R>>3	SEM_R>>3	SHM_R>>3	Группа – чтение
0020	MSG_W>>3	SEM_A>>3	SHM_W>>3	Группа – запись
0004	MSG_R>>6	SEM_R>>6	SHM_R>>6	Прочие – чтение
0002	MSG_W>>6	SEM_A>>6	SHM_W>>6	Прочие – запись

Битам соответствуют поименованные константы, показанные в табл. 3.5. Константы определяются в соответствующих (см. табл. 3.1) заголовочных файлах. Суффикс A в SEM_A означает «alter» (изменение).

Когда процесс предпринимает попытку доступа к объекту IPC, производится двухэтапная проверка – при открытии объекта (функция **getXXX**) и каждый раз при обращении к объекту IPC.

1. При установке доступа к существующему объекту IPC с помощью одной из функций **getXXX** производится первичная проверка аргумента **oflag**. Аргумент не должен указывать биты доступа, не установленные в поле **mode** структуры **ipc_perm** (нижний квадрат на рис. 3.10). Любой процесс, попытавшийся указать эти биты в аргу-

менте **oflag**, получит ошибку. Любой процесс может пропустить эту проверку, указав аргумент **oflag**, равный 0, если заранее известно о существовании объекта IPC.

2. При любой операции с объектами IPC производится проверка разрешений для процесса, эту операцию запрашивающего. Привилегированному пользователю доступ предоставляется всегда.
3. Если действующий идентификатор пользователя совпадает со значением **uid** или **cuid** объекта IPC и установлен соответствующий бит разрешения доступа в поле **mode** объекта IPC, доступ будет разрешен. Под соответствующим битом разрешения доступа подразумевается бит, разрешающий чтение, если вызывающий процесс запрашивает операцию чтения для данного объекта IPC (например, получение сообщения из очереди), или бит, разрешающий запись, если процесс хочет осуществить ее.
4. Если действующий идентификатор группы совпадает со значением **gid** или **cgid** объекта IPC и установлен соответствующий бит разрешения доступа в поле **mode** объекта IPC, доступ будет разрешен.
5. Если доступ не был разрешен на предыдущих этапах, проверяется наличие соответствующих установленных битов доступа для прочих пользователей.

Структура **ipc_perm** содержит переменную **seq**, в которой хранится порядковый номер канала. Эта переменная представляет собой счетчик, заводимый ядром для каждого объекта IPC в системе. При удалении объекта IPC номер канала увеличивается, а при переполнении сбрасывается в ноль. Идентификаторы System V IPC устанавливаются для всей системы, а не для процесса.

Если два неродственных процесса используют, например, одну очередь сообщений, ее идентификатор, возвращаемый функцией **msgget**, должен иметь одно и то же целочисленное значение в обоих процессах, чтобы они получили доступ к одной и той же очереди.

Такая особенность дает возможность процессу, созданному злоумышленником, попытаться прочесть сообщение из очереди, созданной другим приложением, последовательно перебирая различные идентификаторы и надеясь на существование открытой в текущий момент очереди, доступной для чтения всем. Для исключения такой возможности разработчики средств IPC расширили диапазон значений идентификатора путем увеличения значения идентификатора, возвращаемого вызывающему процессу, на количество записей в системной

таблице IPC каждый раз, когда происходит повторное использование одной из них.

Счетчик номеров каналов позволяет исключить повторное использование идентификаторов System V IPC через небольшой срок. Это гарантирует, что досрочно завершивший работу и перезапущенный сервер не станет использовать тот же идентификатор.

Программа slot.c, доступная на сайте, выводит первые десять значений идентификаторов, возвращаемых функцией **msgget**. При очередном прохождении цикла **msgget** создает очередь сообщений, а **msgctl** с командой **IPC_RMID** в качестве аргумента удаляет ее. При повторном запуске программы видим наглядную иллюстрацию того, что последовательный номер канала – это переменная, хранящаяся в ядре и продолжающая существовать и после завершения процесса.

В системах, поддерживающих IPC, предоставляются две специальные утилиты: **ipcs**, выводящая различную информацию о свойствах System V IPC, и **ipcrm**, удаляющая очередь сообщений System V, семафор или сегмент разделяемой памяти. Первая из этих функций поддерживает около десятка параметров командной строки, управляющих отображением информации о различных типах IPC. Второй (**ipcrm**) можно задать до шести параметров. Подробную информацию о них можно получить в справочной системе.

Большинству реализаций System V IPC свойственно наличие внутренних ограничений, налагаемых ядром. Это, например, максимальное количество очередей сообщений или ограничение на максимальное количество семафоров в наборе. Их текущие значения можно вывести на экран, имея права суперпользователя и используя команду **sysctl -a | grep kernel**.

3.2.4. Семафоры System V IPC

Идею управления дорожным движением с помощью семафоров можно без особых изменений перенести на управление доступом к данным. Семафор – особая структура, содержащая число, большее или равное нулю, и управляющая цепочкой процессов, ожидающих особого состояния на данном семафоре. Семафоры могут использоваться для контролирования доступа к ресурсам: число в семафоре представляет собой количество процессов, которые могут получить доступ к данным.

Каждый раз, когда процесс обращается к данным, значение в семафоре должно быть уменьшено на единицу, и увеличено, когда работа с данными будет прекращена. Если ресурс эксклюзивный, то есть к данным должен иметь доступ только один процесс, то начальное значение в семафоре следует установить единицей.

Семафоры можно использовать и для других целей, например для счетчика ресурсов. В этом случае число в семафоре – количество свободных ресурсов (например, количество свободных ячеек памяти).

Рассмотрим реализацию семафоров в System V. Создает семафор функция **semget**:

```
int semget(key_t key, int nsems, int semflg);
```

Здесь **key** – IPC ключ, **nsems** – число семафоров, которое мы хотим создать, и **semflg** – права доступа, закодированные в 12 бит: первые три бита отвечают за режим создания, остальные девять – права на запись и чтение для пользователя, группы и остальных. Более полная информация доступна в **man ipc**.

System V создает сразу несколько семафоров, что уменьшает код. Рассмотрим создание семафора на примере программы semop1.c, доступной для скачивания на сайте в разделе «Semaphores».

Управление происходит с помощью функции **semctl**:

```
int semctl(int semid, int semnum, int cmd, ...);
```

которая выполняет действие **cmd** на наборе семафоров **semid** или на одном семафоре с номером **semnum**. В зависимости от команды может понадобиться указать еще один аргумент следующего типа:

```
union semun {
int val;    /* значение для SETVAL */
struct semid_ds *buf;/*буферы для IPC_STAT, IPC_SET */
unsigned short *array;/* массивы для GETALL, SETALL */
struct seminfo *__buf; /* буфер для IPC_INFO */
};
```

Чтобы изменить значение семафора, используют директиву **SETVAL**, новое значение должно быть указано в **semun**:

```
/* создать только один семафор */
  semid = semget(key, 1, 0666 | IPC_CREAT);
/* в семафоре 0 установить значение 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);
```

Теперь необходимо удалить семафор, освобождая структуры, использовавшиеся для управления им. Это выполняет директива **IPC_RMID**.

Она удаляет семафор и посылает сообщение об этом всем процессам, ожидающим доступа к ресурсу:

```
/* удалить семафор */
  semctl(semid, 0, IPC_RMID);
```

Использовать семафор можно с помощью процедуры semop:

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Здесь **semid** – идентификатор набора семафоров, **sops** – массив, содержащий операции, которые необходимо произвести, **nsops** – число этих операций. Каждая операция представляется структурой **sembuf**:

```
{unsigned short sem_num; short sem_op; short sem_flg;}
```

Операции, которые мы можем указать, являются целыми числами и подчиняются трем правилам.

1. sem_op < 0

Если модуль значения в семафоре больше или равен модулю **sem_op**, то **sem_op** добавляется к значению в семафоре (т.е. значение в семафоре уменьшается). Если модуль **sem_op** больше, то процесс переходит в спящий режим, пока не будет достаточно ресурсов.

2. sem_op = 0

Процесс спит, пока значение в семафоре не достигнет нуля.

3. sem_op > 0

Значение **sem_op** добавляется к значению в семафоре, используемый ресурс освобождается.

Рассмотрим применение семафоров на примере задачи писателей – читателя. Имеется буфер, в который несколько процессов W_1,..., W_n могут писать, и один процесс R может из него читать. Такие операции нельзя выполнять одновременно. Процессы W_i могут писать всегда, когда буфер не полон, а процесс R может читать, когда буфер не пуст. Итак, необходимо три семафора: один управляет доступом к буферу, а два других следят за числом элементов в нем.

Учитывая, что доступ к буферу должен быть эксклюзивным, первый семафор будет бинарным, в то время как второй и третий будут принимать значения, зависящие от размера буфера. Потребность в двух семафорах связана с особенностью работы функции **semop**. Если, например, процессы W_i уменьшают значение в семафоре, отвечающем за свободное место в буфере, до нуля, то процесс R может увеличивать это значение до бесконечности. Поэтому такой семафор не может указывать на отсутствие элементов в буфере.

Текст примера semop2.c доступен на сайте. Прокомментируем наиболее интересные части кода:

```
  struct sembuf lock_res = {0, -1, 0};
  struct sembuf rel_res = {0, 1, 0};
  struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1,
IPC_NOWAIT};
  struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1,
IPC_NOWAIT};
```

Эти четыре строки – действия, которые мы можем производить над семафорами: первые две содержат по одному действию каждая, вторые – по два. Первое действие, **lock_res**, блокирует ресурс, уменьшая значение первого (номер 0) семафора на единицу (если значение в семафоре не нуль), а если ресурс уже занят, то процесс ждет. Действие **rel_res** аналогично **lock_res**, только значение в первом семафоре увеличивается на единицу, т.е. убирается блокировка ресурса.

Действия **push** и **pop** – это массивы из двух действий. Первое действие над семафором номер 1, второе – над семафором номер 2; одно увеличивает значение в семафоре, другое уменьшает, но процесс не будет ждать освобождения ресурса: **IPC_NOWAIT** заставляет его продолжить работу, если ресурс заблокирован.

Далее мы инициализируем значения в семафорах: в первом – единицей, так как он контролирует доступ к ресурсу, во втором – длиной буфера (заданной в командной строке), в третьем – нулем (т.е. числом элементов в буфере).

Затем процесс W_i пытается заблокировать ресурс посредством действия **lock_res**; как только это ему удается, он добавляет элемент в буфер посредством действия **push** и выводит сообщение об этом на стандартный вывод.

Если операция не может быть произведена, процесс выводит сообщение о заполнении буфера. В конце процесс освобождает ресурс.

Процесс R ведет себя практически так же, как и процесс W_i: блокирует ресурс, производит действие **pop**, освобождает ресурс.

Так выглядит работа с семафорами в первом приближении.

3.2.5. Очереди сообщений System V IPC

Каждой очереди сообщений System V сопоставляется свой идентификатор. Любой процесс с соответствующими привилегиями может поместить сообщение в очередь, и любой процесс с соответствующими

привилегиями может сообщение из очереди считать. Для помещения сообщения в очередь System V не требуется наличия подключенного к ней на считывание процесса.

Ядро хранит информацию о каждой очереди сообщений в виде структуры, определенной в заголовочном файле **<sys/msg.h>**:

```
struct msqid_ds {
struct ipc_perm msg_perm;//Разрешения чтения и записи
struct msg *msg_first;/*указатель на первое сообщение
в очереди */
struct msg *msg_last;/* указатель на последнее сообще-
ние в очереди */
msglen_t msg_cbytes;  /* размер очереди в байтах */
msgqnum_t msg_qnum;/*количество сообщений в очереди */
msglen_t msg_qbytes;/* максимальный размер очереди в
байтах */
pid_t msg_lspid; /* pid последнего процесса, вызвавше-
го msgsnd(); */
pid_t msg_lrpid;/* pid последнего msgrcv() */
time_t msg_stime;//время отправки последнего сообщения
time_t msg_rtime;/* время последнего считывания сооб-
щения */
time_t msg_ctime;/* время последнего вызова msgctl(),
изменившего одно из полей структуры */
}
```

Представим конкретную очередь сообщений, хранимую ядром как связный список, на рис. 3.11. В этой очереди три сообщения длиной 1, 2 и 3 байта с типами 100, 200 и 300 соответственно.

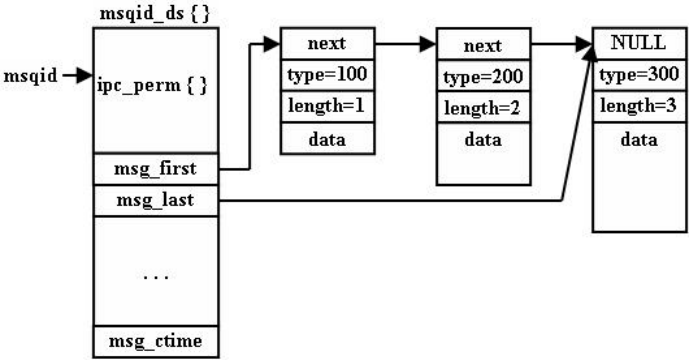

Рис. 3.11. Структура очереди сообщений

Для работы с очередью используется ряд функций. **msgget** создает новую очередь сообщений или получает доступ к существующей:

```
#include <sys/msg.h>
int msgget(key_t key, int oflag);
```

Возвращаемое значение – целочисленный идентификатор, используемый тремя другими функциями **msgXXX** для обращения к данной очереди, или –1 в случае ошибки. Идентификатор вычисляется на основе указанного ключа, который формируется функцией **ftok** или может представлять собой константу **IPC_PRIVATE**.

Флаг **oflag** представляет собой комбинацию разрешений чтения-записи. В него можно добавить флаги **IPC_CREAT** или **IPC_CREAT|IPC_EXCL** с помощью логического сложения. При создании новой очереди инициализируются следующие поля структуры **msqid_ds**:

- полям **uid** и **cuid** структуры **msg_perm** присваивается значение действующего идентификатора пользователя вызвавшего процесса, а полям **gid** и **cgid** – действующего идентификатора группы;
- разрешения чтения-записи, указанные в **oflag**, помещаются в **msg_perm.mode**;
- значения **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime** и **msg_rtime** устанавливаются в 0;
- в **msg_ctime** записывается текущее время;
- в **msg_qbytes** помещается ограничение ядра на размер очереди.

После открытия очереди сообщений с помощью функции **msgget** можно помещать сообщения в эту очередь с помощью **msgsnd**:

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t length,
int flag);
```

Функция возвращает 0 в случае успешного завершения и –1 – в случае ошибки. Здесь **msqid** представляет собой идентификатор очереди, возвращаемый **msgget**. Указатель **ptr** указывает на структуру следующего шаблона, определенного в **<sys/msg.h>**:

```
struct msgbuf {
long mtype; /* тип сообщения, должен быть > 0 */
char mtext[1]; /* данные */
};
```

Тип сообщения (**mtype**) должен быть больше нуля, поскольку неположительные типы используются в качестве специальной команды функции **msgrcv**. Большинство приложений затем определяют собст-

венную структуру сообщений. Например, если приложению нужно передавать сообщения, состоящие из 16-разрядного целого, за которым следует 8-байтовый массив символов, оно может определить свою собственную структуру так:

```
#define MY_DATA 8
typedef struct my_msgbuf {
long mtype; /*тип сообщения */
int16_t mshort; /* начало данных */
char mchar[MY_DATA];
} message;
```

Аргумент **flag** может быть либо 0, либо **IPC_NOWAIT**. В последнем случае он отключает блокировку для **msgsnd**: если для нового сообщения недостаточно места в очереди, возврат из функции происходит немедленно. Это может произойти, если:

- в данной очереди уже имеется слишком много данных (значение **msg_qbytes** в структуре **msqid_ds**);
- во всей системе имеется слишком много сообщений.

Если верно одно из этих условий и установлен флаг **IPC_NOWAIT**, функция **msgsnd** возвращает ошибку с кодом **EAGAIN**. Если флаг **IPC_NOWAIT** не указан, а одно из этих условий выполняется, поток приостанавливается до тех пор, пока:

- для сообщения освободится достаточно места;
- очередь с идентификатором **msqid** будет удалена из системы (в этом случае возвращается ошибка с кодом **EIDRM**);
- вызвавший функцию поток будет прерван перехватываемым сигналом (в этом случае возвращается ошибка с кодом **EINTR**).

Сообщение может быть считано из очереди с помощью **msgrcv**:

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t length,
long type, int flag);
```

Функция возвращает количество данных в сообщении или –1 – в случае ошибки. Аргумент **ptr** указывает, куда следует помещать принимаемые данные. Он указывает на поле данных типа **long**, которое предшествует полезным данным. Аргумент **length** задает размер относящейся к полезным данным части буфера, на который указывает **ptr**. Это максимальное количество данных, которое может быть возвращено функцией. Поле типа **long** не входит в эту длину. Аргумент **type** определяет тип сообщения, которое нужно считать из очереди:

- если тип равно 0, возвращается первое сообщение в очереди;

- если тип больше 0, возвращается первое сообщение, тип которого равен указанному;
- если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше либо равно модулю **type**.

Рассмотрим пример очереди сообщений, изображенный на рис. 3.11. В этой очереди имеются три сообщения:

- первое сообщение имеет тип 100 и длину 1;
- второе сообщение имеет тип 200 и длину 2;
- третье сообщение имеет тип 300 и длину 3.

Табл. 3.6 показывает, какое сообщение будет возвращено при различных значениях аргумента **type**.

Таблица 3.6

Определение типа сообщения по аргументу type

type	Тип возвращаемого сообщения
0	100
100	100
200	200
300	300
–100	100
–200	100
–300	100

Аргумент **flag** указывает что делать, если в очереди нет сообщения с запрошенным типом. Если установлен бит **IPC_NOWAIT**, происходит немедленный возврат из функции **msgrcv** с кодом ошибки **ENOMSG**. В противном случае вызвавший процесс блокируется до тех пор, пока не произойдет одно из следующего:

- появится сообщение с запрошенным типом;
- очередь с идентификатором **msqid** будет удалена из системы (в этом случае будет возвращена ошибка с кодом **EIDRM**);
- вызвавший поток будет прерван перехватываемым сигналом (в этом случае возвращается ошибка **EINTR**).

В аргументе **flag** можно указать бит **MSG_NOERROR**. При установке этого бита данные, превышающие объем буфера (аргумент **length**), будут просто обрезаться до его размера без возвращения кода ошибки. Если этот флаг не указать, при превышении объемом сообщения аргумента **length** будет возвращена ошибка **E2BIG**.

В случае успешного завершения работы **msgrcv** возвращает количество байтов в принятом сообщении. Оно не включает байты, нужные для хранения типа сообщения (**long**), который также возвращается через указатель **ptr**.

Функция **msgctl** позволяет управлять очередями сообщений:

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

Функция возвращает 0 в случае успешного завершения, –1 – в случае ошибки. Команд (аргумент **cmd**) может быть три:

- **IPC_RMID** – удаление очереди с идентификатором **msqid** из системы. Все сообщения, имеющиеся в этой очереди, будут утеряны. Для этой команды третий аргумент функции игнорируется;
- **IPC_SET** – установка значений четырех полей структуры **msqid_ds** данной очереди равными значениям соответствующих полей структуры, на которую указывает аргумент **buff**: **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode**, **msg_qbytes**;
- **IPC_STAT** – возвращает вызвавшему процессу (через **buff**) текущее содержимое структуры **msqid_ds** для очереди **msqid**.

Программа ctl.c, доступная на сайте в папке «SysVmsg», создает очередь сообщений, помещает в нее сообщение с 1 байтом информации, вызывает функцию **msgctl** с командой **IPC_STAT**, выполняет команду **ipcs**, используя функцию **system**, а затем удаляет очередь, вызвав функцию **msgctl** с командой **IPC_RMID**.

Поскольку очереди сообщений System V обладают живучестью ядра, мы можем написать несколько отдельных программ для работы с этими очередями и изучить их действие.

Программа msgcreate.c, доступная на сайте, создает очередь сообщений. Параметр командной строки -e позволяет указать флаг **IPC_EXCL**. Полное имя файла, являющееся обязательным аргументом командной строки, передается функции **ftok**. Получаемый ключ преобразуется в идентификатор функцией **msgget**.

Программа msgsnd.c помещает в очередь одно сообщение заданной длины и типа. В ней создается указатель на структуру **msgbuf**

общего вида, а затем выделяется место под буфер записи соответствующего размера вызовом **calloc**. Эта функция инициализирует буфер нулем.

Программа msgrcv.c считывает сообщение из очереди. В командной строке может быть указан параметр -n, отключающий блокировку, а параметр -t может быть использован для указания типа сообщения в функции **msgrcv**.

Для удаления очереди сообщений мы вызываем функцию **msgctl** с командой **IPC_RMID**, как показано в программе msgrimd.c.

Воспользуемся этими четырьмя программами. Создадим очередь и поместим в нее три сообщения. Файл, указываемый в качестве аргумента **ftok**, обязательно должен существовать. Используем имя существующего файла при создании очереди сообщений. После этого в очередь поместим три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300 (вспомните рис. 3.11). Программа **ipcs** показывает, что в очереди находятся три сообщения общим объемом 6 байт.

Продемонстрируем использование аргумента **type** при вызове **msgrcv** для считывания сообщений в произвольном порядке. Вначале запросим сообщение с типом 200, потом сообщение с наименьшим значением типа, не превышающим 300 (–300), а потом – первое сообщение в очереди (без указания типа). Запуск msgrcv при пустой очереди иллюстрирует действие флага **IPC_NOWAIT**. Если указать положительное значение типа, а сообщений с таким типом в очереди нет, и флаг **IPC_NOWAIT** не установлен, то программа зависнет.

После этого очередь можно удалить с помощью программы msgrmid.c или с помощью системной команды **ipcrm -q [msgid]**.

Покажем, что для получения доступа к очереди сообщений System V не обязательно вызывать **msgget**: все, что нужно, – это знать идентификатор очереди сообщений, который легко получить с помощью **ipcs**, и считать разрешения доступа для очереди. msgrcvid.c есть упрощенный вариант программы msgrcv.c. Здесь используется идентификатор очереди, являющийся аргументом командной строки.

Создадим очередь, запишем в нее сообщение, определим идентификатор с помощью **ipcs** и предоставим его программе msgrcvid в качестве аргумента командной строки. Удаление очереди и ранее выполнялось по идентификатору.

Рассмотрим более сложный пример программы типа клиент-сервер (папка «SysVmsg/cliserv» на сайте) с использованием двух очередей

сообщений. Одна из очередей предназначена для передачи сообщений от клиента серверу, а другая – в обратную сторону.

Программа использует заголовочный файл svmsg.h. В нем подключается заголовочный файл mesg.h и определяются ключи для каждой из очередей сообщений. Программа server_main.c создает обе очереди сообщений, не указывая флаг **IPC_EXCL**. Функция server.c вызывает функции-обертки mesg_send.c и mesg_recv.c. Программа client_main.c открывает две очереди сообщений и вызывает функцию client из файла client.c. Она тоже использует mesg_send.c и mesg_recv.c.

Для демонстрации работы программ нужна еще одна консоль. В ней запустим программу сервер, которая получает от клиента полное имя файла. Если файл не существует, формируется диагностическое сообщение и отправляется клиенту. Если файл существует, он открывается и его содержимое пересылается клиенту. В первой консоли запустим программу клиент. Введем с клавиатуры полное имя файла. Если его не существует, клиент получит и выведет диагностическое сообщение, а если существует – получит и выведет его содержимое.

Наличие поля **type** у каждого сообщения в очереди предоставляет интересные возможности:

1. Поле **type** может использоваться для идентификации сообщений, позволяя нескольким процессам мультиплексировать сообщения в одной очереди. Например, все сообщения от клиентов серверу имеют одно и то же значение типа, тогда как сообщения сервера клиентам имеют различные типы, уникальные для каждого клиента.
2. Поле **type** может использоваться для установки приоритета сообщений. Очереди System V позволяют считывать сообщения в произвольном порядке в зависимости от значений типа сообщений.
3. Можно вызывать **msgrcv** с флагом **IPC_NOWAIT** для считывания сообщений с конкретным типом и немедленного возвращения управления процессу в случае отсутствия таких сообщений.

Рассмотрим такой вариант: один сервер и несколько клиентов (папка «SysVmsg/mpx1q» на сайте). Можно использовать значение типа 1 как тип сообщений от любого клиента серверу. Если клиент передаст серверу свой PID в сообщении, сервер сможет отсылать клиенту сообщения, используя его PID в качестве значения типа сообщения.

На рис. 3.12 приведен пример использования очереди для мультиплексирования сообщений между несколькими клиентами и сервером.

Рис. 3.12. Мультиплексирование сообщений между несколькими клиентами и сервером

В файле server_main.c, папка mpx1q, приведен текст функции main сервера. Заголовочные и вспомогательные файлы остались прежними. Создается единственная очередь сообщений (если она существует, ошибки не возникнет). Идентификатор очереди сообщений используется в качестве обоих аргументов при вызове функции server.

Функция server из файла server.c в той же папке обеспечивает работу сервера. PID процесса, отправляемый клиентом, используется в качестве типа для всех сообщений, отправляемых сервером этому клиенту. Функция представляет собой бесконечный цикл, в котором считываются запросы клиентов и отсылаются запрошенные файлы. Этот сервер является последовательным.

В файле client_main.c приведен текст функции main клиента. Клиент открывает очередь сообщений, которая должна была быть создана сервером заранее. Функция client, текст которой дан в файле client.c, обеспечивает всю обработку со стороны клиента. Тип сообщений, запрашиваемых функцией mesg_recv, совпадает с PID процесса клиента.

Запуск программ производится так же, как в предыдущем примере.

Изменим пример так, чтобы запросы клиентов передавались по одной очереди, а для ответов использовалась бы отдельная очередь для каждого клиента. На рис. 3.13 изображена структура приложения.

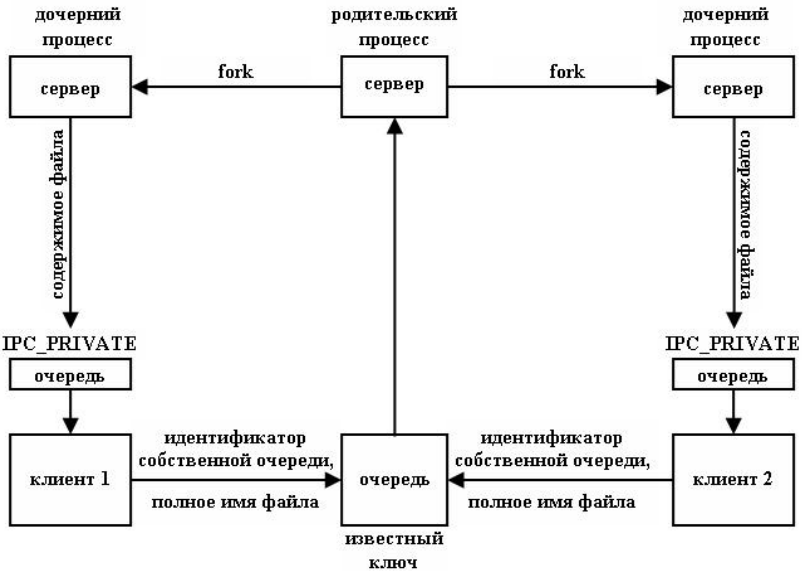

Рис. 3.13. Мультиплексирование сообщений с параллельным сервером

Ключ очереди сервера должен быть известен клиентам, а сами клиенты создают свои очереди с ключом **`IPC_PRIVATE`**. Вместо передачи серверу PID процесса клиенты сообщают ему идентификатор своей очереди, в которую сервер направляет свой ответ. Этот сервер является параллельным: для каждого нового клиента порождается отдельный процесс. При такой схеме может возникнуть проблема в случае «гибели» клиента, ибо тогда сообщения останутся в его очереди до перезагрузки ядра или явного удаления очереди другим процессом.

Заголовочные файлы mesg.h, svmsg.h, функции main сервера mesg send не претерпевают изменений по сравнению с предыдущими версиями. Функция main клиента приведена в файле client_main.c (папка «SysVmsg/mpx2q» на сайте); она слегка изменилась. Очередь сервера открывается с известным ключом (MQ_KEY1), а собственная очередь создается с ключом **`IPC_PRIVATE`**. Два идентификатора этих очередей становятся аргументами функции client. После завершения работы клиента его персональная очередь удаляется.

Функция client практически идентична предыдущей, но вместо передачи идентификатора процесса клиента на сервер направляется иден-

тификатор очереди клиента. Тип сообщения в структуре **mesg** остается равным единице, поскольку это значение устанавливается для сообщений, передаваемых в обе стороны.

Главное отличие функции server от предыдущего примера в том, что эта функция представляет собой бесконечный цикл, в котором для каждого нового клиента вызывается **fork**. Поскольку для каждого клиента порождается отдельный процесс, нужно позаботиться о процессах-зомби. Установим обработчик для сигнала **SIGCHLD**, и функция **sig_chld** (файл sigchldwaitpid.c) будет вызываться при завершении работы дочернего процесса.

Вызовом **fork** порождается новый процесс, который блокируется в вызове mesg_recv, ожидая появления сообщения от очередного клиента, производит попытку открыть запрошенный файл и отправляет клиенту либо сообщение об ошибке, либо содержимое файла.

Вызов **fopen** преднамеренно помещен в дочерний процесс, а не в родительский, поскольку если файл находится в удаленной файловой системе, его открытие может занять довольно много времени.

Запуск программ производится так же, как в предыдущих примерах.

В табл. 3.7 приведены значения системных ограничений для двух разных реализаций ОС. Первая колонка представляет собой традиционное имя System V для переменной ядра, хранящей это ограничение.

В ОС Linux параметры ядра заданы в файле **/etc/sysctl.conf** в секции #Controls message queues. Определить ограничения на очереди при наличии прав суперпользователя можно командой **sysctl -a | grep kernel.msg.**

Таблица 3.7

Системные ограничения на очереди сообщений

Имя	Описание	ASPLinux 9.0	Solaris 2.6
msgmax	Максимальное количество байтов в сообщении	8192	2048
msgmnb	Максимальное количество байтов в очереди сообщений	16384	4096
msgmni	Максимальное количество очередей сообщений в системе	16	50

В файле limits.c приведен текст программы, которая определяет ограничения, показанные в табл. 3.7, для текущей системы.

Для определения максимально возможного размера сообщения она пытается послать сообщение, в котором будет 65 536 байт данных, и если эта попытка оказывается неудачной, уменьшает этот объем до 65 408, и т. д., пока вызов **msgsnd** не окажется успешным.

Создавая 8-байтовые сообщения, программа смотрит, сколько их поместится в очередь. После определения этого ограничения очередь удаляется и повторяется процедура с 16-байтовыми сообщениями, пока не будет достигнут максимальный размер сообщения.

Системное ограничение на количество одновременно открытых идентификаторов определяется непосредственно созданием очередей до тех пор, пока не произойдет ошибка при вызове **msgget**.

3.2.6. Разделяемая память System V IPC

Разделяемая память является самым быстрым средством межпроцессного взаимодействия. После отображения области памяти в адресное пространство процессов, совместно ее использующих, для передачи данных между процессами не требуется участие ядра. Обычно, однако, требуется некоторая форма синхронизации процессов, помещающих данные в разделяемую память и считывающих ее оттуда.

Рассмотрим работу программы чтения файла типа клиент-сервер, пример для иллюстрации различных способов передачи сообщений.

Сервер считывает данные из входного файла. Данные из файла считываются в ядро, а затем копируются из ядра в память процесса. Сервер составляет сообщение из этих данных и отправляет его, используя именованный или неименованный канал или очередь сообщений.

Клиент считывает данные из канала IPC, что обычно требует их копирования из ядра в пространство процесса. Наконец, данные копируются из буфера клиента в выходной файл.

Для передачи содержимого файла требуются четыре операции копирования данных. Эти операции копирования между процессами и ядром являются дорогостоящими (более дорогостоящими, чем копирование данных внутри ядра или одного процесса). На рис. 3.14 изображено перемещение данных между клиентом и сервером через ядро.

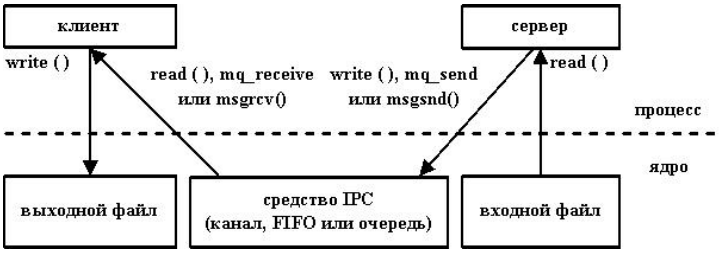

Рис. 3.14. Передача содержимого файла через ядро

Недостаток этих форм IPC в том, что для передачи между процессами информация должна пройти через ядро. Разделяемая память дает возможность обойти этот недостаток, поскольку ее использование позволяет двум процессам обмениваться данными через общий участок памяти. Одновременное использование участка памяти во многом аналогично совместному доступу к файлу.

Теперь информация передается между клиентом и сервером в такой последовательности:

- сервер получает доступ к объекту разделяемой памяти, используя для синхронизации семафор (например);
- сервер считывает данные из файла в разделяемую память. Второй аргумент вызова read указывает на объект разделяемой памяти;
- после завершения операции считывания клиент уведомляется сервером с помощью семафора;
- клиент записывает данные из объекта разделяемой памяти в выходной файл.

Этот сценарий иллюстрирует рис. 3.15.

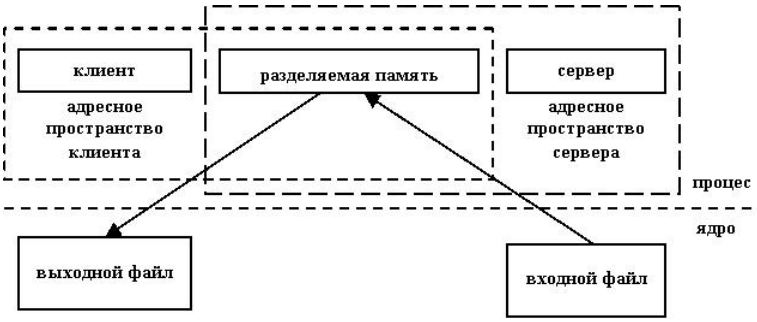

Рис. 3.15. Передача содержимого файла через разделяемую память

Из рисунка видно, что копирование данных происходит лишь дважды: из входного файла в разделяемую память и из разделяемой памяти в выходной файл. Два прямоугольника штриховыми линиями подчеркивают, что разделяемая память принадлежит как адресному пространству клиента, так и адресному пространству сервера.

Основные принципы разделяемой памяти System V совпадают с концепцией разделяемой памяти Posix. Вместо вызовов **shm_open** и **imap** в Posix здесь используются вызовы **shmget** и **shmat**.

Для каждого сегмента разделяемой памяти ядро хранит структуру **shimd_ds**, определенную в заголовочном файле **<sys/shm.h>**:

```
struct shimd_ds {
struct ipc_perm shm_perm; /* структура разрешений */
size_t shm_segsz; /* размер сегмента */
pid_t shm_lpid; /* идентификатор последнего процесса*/
pid_t shm_cpid; /*идентификатор процесса-создателя*/
shmatt_t shm_nattch;/*текущее количество подключений*/
shmat_t shm_cnattch; // количество подключений in-core
time_t shm_atime; /* время последнего подключения */
time_t shm_dtime; /* время последнего отключения */
time_t shm_ctime; /*время последнего изменения */
};
```

Структура **ipc_perm** содержит разрешения доступа к сегменту разделяемой памяти.

С помощью функции **shmget** можно создать новый сегмент разделяемой памяти или подключиться к существующему:

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int oflag);
```

Возвращаемое целочисленное значение называется идентификатором разделяемой памяти. Он используется с тремя другими функциями **shmXXX**. В случае ошибки возвращается –1.

Аргумент **key** может содержать значение, возвращаемое функцией **ftok**, или константу **IPC_PRIVATE**. Аргумент **size** указывает размер сегмента в байтах. При создании нового сегмента разделяемой памяти нужно указать ненулевой размер. Если производится обращение к существующему сегменту, аргумент **size** должен быть нулевым.

Флаг **oflag** представляет собой комбинацию флагов доступа на чтение и запись. К ним могут быть добавлены с помощью логического сложения флаги **IPC_CREAT** или **IPC_CREAT | IPC_EXCL**. Новый сегмент инициализируется нулями.

Функция **shmget** создает или открывает сегмент разделяемой памяти, но не дает вызвавшему процессу доступа к нему. Для подключения сегмента разделяемой памяти к адресному пространству процесса предназначена функция **shmat**:

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int flag);
```

Аргумент **shmid** – это идентификатор разделяемой памяти, возвращенный **shmget**. Функция **shmat** возвращает адрес начала области разделяемой памяти в адресном пространстве вызвавшего процесса или –1 в случае ошибки. Правила формирования адреса таковы:

- если аргумент **shmaddr** представляет собой нулевой указатель, система сама выбирает начальный адрес. Это рекомендуемый (и обеспечивающий наилучшую совместимость) метод;
- если **shmaddr** отличен от нуля, возвращаемый адрес зависит от того, был ли указан флаг **SHM_RND** (в аргументе **flag**);
- если флаг **SHM_RND** не указан, разделяемая память подключается непосредственно с адреса, указанного аргументом **shmaddr** ;
- если флаг **SHM_RND** указан, сегмент разделяемой памяти подключается с адреса, указанного аргументом **shmaddr**, округленного вниз до кратного константе **SHMLBA**. Аббревиатура LBA означает lower boundary address – нижний граничный адрес.

По умолчанию, при наличии разрешений, сегмент подключается для чтения и записи. В аргументе **flag** можно указать константу **SHM_RDONLY**, которая позволит установить доступ только на чтение.

После завершения работы с сегментом разделяемой памяти его следует отключить вызовом **shmdt**:

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Функция возвращает 0 в случае успешного завершения, –1 – в случае ошибки. Эта функция не удаляет сегмент разделяемой памяти. Удаление осуществляется функцией **shmctl** с командой **IPC_RMID**. При завершении работы процесса все сегменты, которые не были отключены им явно, отключаются автоматически, но не удаляются.

Функция **shmctl** позволяет выполнять различные операции с сегментом разделяемой памяти:

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

Функция возвращает 0 в случае успешного завершения, –1 в случае ошибки. Команд (значений аргумента **cmd**) может быть три:

1) **IPC_RMID** – удаление сегмента разделяемой памяти с идентификатором **shmid** из системы;
2) **IPC_SET** – установка значений полей структуры **shmid_ds** для сегмента разделяемой памяти равными значениям соответствующих полей структуры, на которую указывает аргумент **buff**: **shm_perm.uid**, **shm_perm.gid**, **shm_perm.mode**. Значение поля **shm_ctime** устанавливается равным текущему системному времени;
3) **IPC_STAT** – возвращает (через аргумент **buff**) текущее значение структуры **shmid_ds** для указанного сегмента разделяемой памяти.

Приведем несколько примеров простых программ, иллюстрирующих работу с разделяемой памятью System V.

Программа shmget.c, текст которой доступен на сайте в разделе «SysVshm», создает сегмент разделяемой памяти, принимая из командной строки полное имя и длину сегмента.

Вызов **shmget** создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ IPC System V вызовом **ftok**. Если указан параметр -e, наличие существующего сегмента с тем же именем приведет к возвращению ошибки. Если мы знаем, что сегмент уже существует, в командной строке должна быть указана нулевая длина. После этого программа завершает работу. Разделяемая память System V обладает живучестью ядра, поэтому сегмент не удаляется.

Тривиальная программа shmrmid.c вызывает **shmctl** с командой **IPC_RMID** для удаления сегмента разделяемой памяти из системы.

Программа shmwrite.c заполняет сегмент разделяемой памяти последовательностью значений 0,1,2,…,254,255,0,1… Сегмент открывается вызовом **shmget** и подключается вызовом **shmat**. Его размер может быть получен вызовом **shmctl** с командой **IPC_STAT**.

Программа shmread.c проверяет последовательность значений, записанную в разделяемую память программой shmwrite.c.

Проиллюстрируем применение этих простых программ. Создадим в системе ASPlinux сегмент разделяемой памяти длиной 1234 байта. Для идентификации сегмента используем полное имя исполняемого файла shmget. Это имя будет передано функции **ftok**. Имя исполняемого файла часто используется в качестве уникального идентификатора:

```
[root@gun_linux_vm]# ./shmget shmget 1234
[root@gun_linux_vm]# ipcs -m
```

Программу **ipcs** запускаем для того, чтобы убедиться, что сегмент разделяемой памяти был создан и не был удален по завершении программы shmget. Количество подключений равно нулю.

Запустим программу shmwrite, чтобы заполнить содержимое разделяемой памяти последовательностью значений. Проверим содержимое сегмента разделяемой памяти программой shmread и удалим сегмент:

```
[root@gun_linux_vm]# ./shmwrite shmget
[root@gun_linux_vm]# ./shmread shmget
[root@gun_linux_vm]# ./shmrmid shmget
[root@gun_linux_vm]# ipcs -m
```

Программа **ipcs** позволит убедиться, что сегмент разделяемой памяти действительно был удален.

В табл. 3.8 приведены значения ограничений ядра для разных реализаций. В первом столбце приведены традиционные для System V имена переменных ядра, в которых хранятся эти ограничения.

Таблица 3.8

Системные ограничения на разделяемую память

Имя	Описание	DUnix 4.0	Solaris 2.6
shmmax	Максимальный размер сегмента, байт	4 194 304	1 048 576
shmmnb	Минимальный размер сегмента, байт	1	1
shmmni	Максимальное количество идентификаторов в системе	128	100
shmseg	Максимальное количество сегментов, подключенных к процессу	32	6

Программа limits.c определяет значения четырех ограничений, приведенных в табл. 3.8. Запустив эту программу в ASPlinux, увидим:

```
[root@gun_linux_vm]# ./limits
4096 identifiers open at once
4096 shared memory segments attached at once
minimum size of shared memory segment=1
max size of shared memory segment=33554432
```

Эти ограничения при наличии прав суперпользователя можно проверить командой **sysctl -a | grep kernel.shm***.

3.4. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В Linux

Поддержка потоков в UNIX появилась с принятием стандарта POSIX. Согласно нему поток создается при помощи вызова:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_
attr_t *attr,void* (*start)(void *),void *arg);
```

Упрощенно вызов **pthread_create(&thr,NULL,start,NULL)** создаст поток, который начнет выполнять функцию **start** и запишет в переменную **thr** идентификатор созданного потока.

Первый аргумент этой функции **thr** – это указатель на переменную типа **pthread_t**, в которую будет записан идентификатор созданного потока, который впоследствии можно будет передавать другим вызовам, когда потребуется сделать что-либо с этим потоком.

Здесь наблюдается особенность POSIX API, а именно непрозрачность базовых типов. Дело в том, что практически ничего нельзя сказать про тип **pthread_t**. Единственное, о чем сказано в стандарте: эти значения можно копировать и, используя вызов **int pthread_equal (pthread_t thr1, pthread_t thr2)**, можно установить, что оба идентификатора **thr1** и **thr2** идентифицируют один и тот же поток (при этом они вполне могут быть неравны в смысле оператора равенства).

Второй аргумент функции **pthread_create, attr** – указатель на переменную типа **pthread_attr_t**, которая задает набор свойств создаваемого потока. Это вторая особенность POSIX API, а именно концепция атрибутов. Дело в том, что в этом API во всех случаях, когда при создании или инициализации некоторого объекта необходимо задать набор неких дополнительных его свойств, вместо указания этого набора при помощи набора параметров вызова используется передача предварительно сконструированного объекта набора атрибутов.

Такое решение имеет, по крайней мере, два преимущества. Во-первых, можно зафиксировать набор параметров функции без угрозы его изменения в дальнейшем, когда у этого объекта появятся новые свойства. Во-вторых, можно многократно использовать один и тот же набор атрибутов для создания множества объектов.

Третий аргумент вызова **pthread_create** – это указатель на функцию типа **void* ()(void *)**. Именно эту функцию и начинает

выполнять вновь созданный поток, при этом в качестве параметра этой функции передается четвертый аргумент вызова **pthread_create**. Таким образом, можно с одной стороны параметризовать создаваемый поток кодом, который он будет выполнять, с другой стороны – параметризовать его различными данными, передаваемыми коду.

Функция **pthread_create** возвращает нулевое значение в случае успеха и ненулевой код ошибки – в случае неудачи. Это также одна из особенностей POSIX API, вместо стандартного для Unix подхода, когда функция возвращает лишь некоторый индикатор ошибки, а код ошибки устанавливает в переменной **errno**. Функции POSIX API возвращают код ошибки в результате своего аргумента.

В качестве резюме рассмотрим пример, thread1.c, который можно скачать с сайта в папке «Threads».

Хотя функции работы с потоками описаны в файле включения **pthread.h**, на самом деле они находятся в библиотеке **libgcc.a**. Поэтому процесс компиляции и сборки многопоточной программы выполняется в два этапа:

```
gcc -Wall -c -o test.o test.c
gcc -Wall -o test test.o <path>libgcc.a -lpthread
```

В большинстве версий Linux библиотека лежит в **/usr/lib/**.

3.4.1. Завершение потоков

Поток завершается, когда происходит возврат из функции. Если нужно получить возвращаемое этой функцией значение, надо воспользоваться такой функцией:

```
int pthread_join(pthread_t thread, void** value_ptr);
```

Эта функция дожидается завершения потока с идентификатором **thread** и записывает возвращаемое ею значение в переменную, на которую указывает **value_ptr**. При этом освобождаются все ресурсы, связанные с потоком, потому эта функция может быть вызвана для данного потока только один раз.

Если возврат значения через **pthread_join** не удобен, например, необходимо получить данные из нескольких потоков, то следует воспользоваться каким-либо другим механизмом, например, организовать очередь возвращаемых значений или возвращать значение в структуре, указатель на которую передают в качестве параметра потока. То есть использование **pthread_join** – это вопрос удобства, а не догма, в отличие от случая пары **fork() – wait()**.

В том случае, если нужно использовать другой механизм возврата или возвращаемое значение просто не интересует, то можно отсоединить (**detach**) поток, сказав тем самым, что нужно освободить ресурсы, связанные с потоком, сразу по завершению функции потока. Сделать это можно несколькими способами.

Во-первых, можно сразу создать поток отсоединенным, задав соответствующий объект атрибутов при вызове **pthread_create**.

Во-вторых, любой поток можно отсоединить, вызвав в любой момент его жизни (то есть до вызова **pthread_join()**) функцию **int pthread_detach(pthread_t thread)** и указав ей в качестве параметра идентификатор потока. При этом поток вполне может отсоединить сам себя, получив свой идентификатор при помощи функции **pthread_t pthread_self(void)**. Следует подчеркнуть, что отсоединение потока никоим образом не влияет на процесс его выполнения, а просто помечает поток как готовый по своему завершению к освобождению ресурсов.

Под освобождаемыми ресурсами подразумеваются в первую очередь стек, память, в которую сохраняется контекст потока, данные, специфичные для потока и тому подобное. Сюда не входят ресурсы, выделяемые явно, например, память, выделяемая через **malloc**, или открываемые файлы. Подобные ресурсы следует освобождать явно.

Помимо возврата из функции потока существует вызов, аналогичный вызову **exit()** для процессов:

```
int pthread_exit(void *value_ptr);
```

Этот вызов завершает выполняемый поток, возвращая в качестве результата его выполнения **value_ptr**. При вызове этой функции поток из нее просто не возвращается. Нужно помнить, что функция **exit()** по-прежнему завершает процесс, то есть уничтожает все потоки.

Рассмотрим соответствующий пример, thread2.c.

Для досрочного завершения потока можно воспользоваться функцией **pthread_cancel(pthread_t thread)**. Единственным ее аргументом является идентификатор потока **thread**. Функция **pthread_cancel()** возвращает 0 в случае успеха и ненулевое значение – в случае ошибки. Хотя **pthread_cancel()** может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков, поскольку поток может не только самостоятельно выбрать порядок завершения, но и игнорировать этот вызов. Вызов функции **pthread_cancel()** следует рассматривать как запрос на

выполнение досрочного завершения потока. Функция **pthread_ setcan-celstate()** определяет, будет ли поток реагировать на обращение к нему с помощью **pthread_cancel()**, или не будет. У функции **pthread_setcancelstate()** два параметра: параметр **state** типа **int** и параметр **oldstate** типа ***int**. В первом передается значение, указывающее, как поток должен реагировать на запрос **pthread_ cancel()**, а в переменную, чей адрес был передан во втором параметре, функция записывает прежнее значение. Если прежнее значение не интересует, во втором параметре можно передать NULL.

Чаще всего функция **pthread_setcancelstate()** используется для временного запрета завершения потока. Если в потоке есть участок кода, во время выполнения которого завершать поток крайне нежелательно, то можно оградить этот участок кода от досрочного завершения с помощью пары вызовов **pthread_setcancelstate()**:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
... //Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Первый вызов **pthread_setcancelstate()** запрещает досрочное завершение потока, второй – разрешает. Если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы, выполнение запроса будет отложено до тех пор, пока функция **pthread_setcancelstate()** не будет вызвана с аргументом **PTHREAD_CANCEL_ENABLE**.

Интересна роль функции **pthread_testcancel(void)**. Эта функция создает точку отмены потока. Даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (а этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. В соответствии со стандартом POSIX точками отмены являются вызовы многих «обычных» функций, например **open()**, **pause()** и **write()**. Про функцию **printf()** в документации сказано, что она может быть точкой отмены, но в Linux при попытке остановиться на **printf()** поток завершается, а **pthread_join()** не возвращает управления. Поэтому создается явная точка отмены с помощью вызова **pthread_testcancel()**.

Можно выполнить досрочное завершение потока, не дожидаясь точек останова. Для этого необходимо перевести поток в режим немед-

ленного завершения с помощью вызова **pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL)**. В этом случае беспокоиться о точках останова уже не нужно. Вызов **pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)** снова переводит поток в режим отложенного досрочного завершения.

3.4.2. Особенности главного потока

Программа на Си начинается с выполнения функции **main()**. Поток, в котором выполняется данная функция, называется главным или начальным (так как это первый поток в приложении). Этот поток обладает многими свойствами обычного потока, для него можно получить идентификатор, он может быть отсоединен, для него можно вызвать **pthread_join** из какого-либо другого потока. Но он обладает и некоторыми особенностями, отличающими его от других потоков.

Возврат из этого потока завершает весь процесс. Если не надо, чтобы по завершении этого потока остальные потоки были уничтожены, то следует воспользоваться функцией **pthread_exit**.

У функции этого потока не один параметр типа **void***, как у остальных, а пара **argc-argv**.

Многие реализации отводят на стек начального потока больше памяти, чем на стеки остальных потоков. Это связано с тем, что существует много традиционных однопоточных приложений, требующих значительного объема стека.

3.4.3. Жизненный цикл потоков

Рассмотрим жизненный цикл потока, а именно последовательность состояний, в которых пребывает поток за время своего существования. В целом (табл. 3.9) можно выделить четыре таких состояния.

Любой создаваемый поток начинает свою жизнь в состоянии «готов». После чего в зависимости от политики планирования системы он может либо сразу перейти в состояние «выполняется», либо перейти в него через некоторое время.

Типичной ошибкой считается, что (в отсутствие явных мер по синхронизации потоков) после возврата из функции **pthread_create** новый поток будет существовать. Но при некоторых политиках планирования и атрибутах потока может статься, что новый поток успеет выполниться к моменту возврата из этой функции.

Таблица 3.9

Состояния потока

Состояние потока	Что означает
Готов (Ready)	Поток готов к выполнению, но ожидает процессора. Возможно, он только что был создан, был вытеснен с процессора другим потоком, или только что был разблокирован
Выполняется (Running)	Поток сейчас выполняется. Следует заметить, что на многопроцессорной машине может быть несколько потоков в таком состоянии
Заблокирован (Blocked)	Поток не может выполняться, так как ожидает чего-либо. Например, окончания операции ввода-вывода
Завершен (Terminated)	Поток был завершен, например, вследствие возврата из функции потока, вызова **pthread_exit**. Поток не был отсоединен и для него не была вызвана функция **pthread_ join**. Как только происходит одно из этих событий, поток перестает существовать

3.4.4. Атрибуты потоков

Атрибуты являются способом определить поведение потока, отличное от поведения по умолчанию. При создании потока с помощью **pthread_create()** или при инициализации переменной синхронизации может быть определен собственный объект атрибутов. Атрибуты определяются только во время создания потока; они не могут быть изменены в процессе использования.

Обычно вызываются три функции.

- Инициализация атрибутов потока – **pthread_attr_init()** создает объект **pthread_attr_t tattr** по умолчанию.
- Изменение значений атрибутов (если значения по умолчанию не подходят) – разнообразные функции **pthread_attr_*()**, позволяющие установить значения индивидуальных атрибутов для структуры **pthread_attr_t tattr**.
- Создание потока – вызов **pthread_create()** с соответствующими значениями атрибутов в структуре **pthread_attr_t tattr**.

Рассмотрим пример кода (thr_attr.c), выполняющего эти действия.

Объект атрибутов является закрытым и не может быть изменен операциями присваивания. Как только атрибут инициализируется и конфигурируется, он доступен всему процессу. Поэтому рекомендуется конфигурировать все требуемые спецификации состояния один раз на ранних стадиях выполнения программы. При этом соответствующий объект атрибутов может использоваться везде, где это нужно.

Использование объектов атрибутов имеет два преимущества.

1. Это обеспечивает мобильность кода. Даже в случае, когда поддерживаемые атрибуты могут измениться в зависимости от реализации, не нужно будет изменять вызовы функций, которые создают объекты потоков, потому что объект атрибутов скрыт от интерфейса. Задача портирования облегчается, потому что объекты атрибутов будут инициализироваться однажды и в определенном месте.
2. Упрощается спецификация состояний в приложении. Пусть в процессе существует несколько множеств потоков, при этом каждое обеспечивает отдельный сервис и имеет свои собственные требования к состоянию. В некоторый момент на ранних стадиях приложения можно инициализировать объект атрибутов потока для каждого множества. Все будущие вызовы создания потока будут обращаться к объекту атрибутов, инициализированному для этого типа потока.

Функция **pthread_attr_init()** используется, чтобы инициализировать объект атрибутов значениями по умолчанию. Память распределяется системой потоков во время выполнения.

Пример вызова функции:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
ret = pthread_attr_init(&tattr);
```

Функция возвращает 0 после успешного завершения. Любое другое значение указывает, что произошла ошибка. Код ошибки устанавливается в переменной **errno**.

Значения по умолчанию для атрибутов приведены в табл. 3.10.

Таблица 3.10

Атрибуты потоков

Атрибут	Значение по умолчанию	Назначение
detach-state	**PTHREAD_CREATE_JOINABLE**	Управление состоянием потока (присоединяемый **PTHREAD_CREATE_JOINABLE** /отсоединяемый **PTHREAD_CREATE_DEACHED**)
sched-policy	**SCHED_OTHER**	Выбор политики диспетчеризации: **SCHED_OTHER** (non-realtime), **SCHED_RR** (realtime) или **SCHED_FIFO** (realtime)
sched-param	0	Приоритет при диспетчеризации имеет смысл только для **SCHED_RR** и **SCHED_FIFO**
inher-itsched	**PTHREAD_EXPLICIT_SCHED**	Параметры диспетчеризации задаются или наследуются от родительского потока (**PTHREAD_ INHERIT_SCHED**)
scope	**PTHREAD_SCOPE_SYSTEM**.	**PTHREAD_SCOPE_SYSTEM** – поток конкурирует за процессор со всеми потоками системы. **PTHREAD_SCOPE_PROCESS** – поток конкурирует за процессор с потоками, созданными родительским потоком

Функция **pthread_attr_destroy()** используется, чтобы удалить память для атрибутов, выделенную во время инициализации. Объект атрибутов становится недействительным.

```
ret = pthread_attr_init(&tattr);
. . .
ret = pthread_attr_destroy(&tattr);
```

Функция возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

Если поток создается отделенным (**PTHREAD_CREATE_DETACHED**), его PID и другие ресурсы могут использоваться, как только он завер-

шится. Для этого можно вызвать перед его созданием функцию **pthread_attr_setdetachstate()**. По умолчанию поток создается неотделенным (**PTHREAD_CREATE_JOINABLE**), и предполагается, что создающий поток будет ожидать его завершения и выполнять **pthread_join()**. Независимо от типа потока процесс не закончится, пока не завершатся все потоки. **pthread_attr_setdetachstate()** возвращает 0 после успешного завершения или другое значение в случае ошибки.

```
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Функция **pthread_attr_getdetachstate()** позволяет определить состояние при создании потока, то есть был ли он отделенным или присоединяемым. Она возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

```
pthread_attr_getdetachstate (&tattr, &detachstate);
```

Поток может быть неограничен (имеет тип **PTHREAD_SCOPE_SYSTEM**) или ограничен (имеет тип **PTHREAD_SCOPE_PROCESS**). Оба этих типа доступны только в пределах данного процесса. Функция **pthread_attr_setscope()** позволяет создать потоки указанных типов. **pthread_attr_setscope()** возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

Функция **pthread_attr_getscope()** используется для определения ограниченности потока. **pthread_att_getscope()** возвращает 0 после успешного завершения или другое значение в случае ошибки.

```
ret = pthread_attr_getscope (&tattr, &scope);
```

Стандарт POSIX определяет значения атрибута планирования: **SCHED_FIFO**, **SCHED_RR** (Round Robin), или **SCHED_OTHER** (метод приложения). Дисциплины **SCHED_FIFO** и **SCHED_RR** поддерживаются только для потоков в режиме реального времени. Попытка установить их в других режимах приведет к возникновению ошибки **ENOSUP**.

```
ret=pthread_attr_setschedpolicy (&tattr, SCHED_OTHER);
```

Парная функция **pthread_attr_getschedpolicy()**, которая возвращает константу, определяющую дисциплину диспетчеризации.

Функция **pthread_attr_setinheritsched()** используется для наследования дисциплины диспетчеризации из родительского потока. Если атрибут **inherit = PTHREAD_ INHERIT_SCHED** (по умолчанию), то будет использована дисциплина планирования родителя. Если

используется константа **PTHREAD_EXPLICIT_SCHED**, используются атрибуты, переданные в вызове **pthread_create()**. Функция возвращает 0 при успешном завершении, и любое другое значение в случае ошибки.

```
pthread_attr_setinheritsched(&tattr,PTHREAD_EXPLICIT_SCHED);
```

Функцию **pthread_attr_getinheritsched (&tattr, int *inherit)** можно использовать для получения информации о дисциплине планирования текущего потока.

Параметры диспетчеризации определены в структуре **sched_param**; поддерживается только приоритет **sched_param.sched_priority**. Этот приоритет задается целым числом и чем выше значение, тем выше приоритет потока при планировании. Создаваемые потоки получают этот приоритет. Функция **pthread_attr_setschedparam()** используется, чтобы установить значения в этой структуре. При успешном завершении она возвращает 0.

```
sched_param param;
param.sched_priority = 20;
ret = pthread_attr_setschedparam (&tattr, &param);
```

Функция **pthread_attr_getschedparam (pthread_attr_t *tattr, const struct sched_param *param)** используется для получения приоритета текущего потока.

3.5. СРЕДСТВА СИНХРОНИЗАЦИИ ПОТОКОВ В Linux

Взаимные исключения (mutual exclusion – mutex) и *условные переменные* (conditional variables) являются основными средствами синхронизации действий нескольких программных потоков или процессов. Обычно это требуется для предоставления нескольким потокам или процессам совместного доступа к данным.

Взаимные исключения и условные переменные появились в стандарте Posix.l для программных потоков и всегда могут быть использованы для синхронизации отдельных потоков одного процесса. Стандарт Posix также разрешает использовать взаимное исключение или условную переменную и для синхронизации нескольких процессов, если это исключение или переменная хранится в области памяти, совместно используемой процессами.

Применение взаимных исключений и условных переменных иллюстрируется классической задачей производитель–потребитель. В примере используются программные потоки, а не процессы, поскольку предоставить потокам общий буфер данных, предполагаемый в этой задаче, легко, а вот создать буфер данных между процессами можно только с помощью одной из форм разделяемой памяти. Другое решение этой задачи возможно с использованием семафоров.

Взаимное исключение (mutex) является простейшей формой синхронизации. Оно используется для защиты *критической области* (critical region), предотвращая одновременное выполнение участка кода несколькими потоками или процессами:

- блокировать_mutex(…);
- критическая область
- разблокировать_mutex(…);

Поскольку только один поток может заблокировать взаимное исключение в любой момент времени, это гарантирует, что только один поток будет выполнять код, относящийся к критической области.

Взаимные исключения по стандарту Posix объявлены как переменные с типом **pthread_mutex_t**. Если переменная-исключение выделяется статически, ее можно инициализировать константой **PTHREAD_MUTEX_INITIALIZER**:

```
static pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
```

При динамическом выделении памяти под взаимное исключение (например, вызовом **malloc**) или при помещении его в разделяемую память необходимо инициализировать эту переменную во время выполнения, вызвав функцию **pthread_mutex_init**.

Следующие три функции используются для установки и снятия блокировки взаимного исключения:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_trylock (pthread_mutex_t *mptr);
int pthread_mutex_unlock (pthread_mutex_t *mptr);
```

Все три возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

При попытке заблокировать взаимное исключение, которое уже заблокировано другим потоком, функция **pthread_mutex_lock** будет ожидать его разблокирования, а **pthread_mutex_trylock** (неблокируемая функция) вернет ошибку с кодом **BUSY**.

Взаимное исключение обычно используется для защиты совместно используемых несколькими потоками или процессами данных и представляет собой блокировку *коллективного пользования*. Это значит, что все потоки, работающие с данными, должны блокировать взаимное исключение. Ничто не может помешать потоку работать с данными, не заблокировав взаимное исключение. Взаимные исключения предполагают добровольное сотрудничество потоков.

Одна из классических задач на синхронизацию называется задачей производителей и потребителей (писателей и читателей). Она также известна как задача ограниченного буфера. Один или несколько производителей (потоков или процессов) создают данные, которые обрабатываются одним или несколькими потребителями. Эти данные передаются между производителями и потребителями с помощью одной из форм IPC. Схема задачи изображена на рисунке.

В одном процессе у нас имеется несколько потоков-производителей и один поток-потребитель. Целочисленный массив **`buff`** содержит производимые и потребляемые данные (данные совместного пользования) Для простоты производители просто устанавливают значение **`buff[0]`** в 0, **`buff[1]`** в 1 и т. д. Потребитель перебирает элементы массива, проверяя правильность записей.

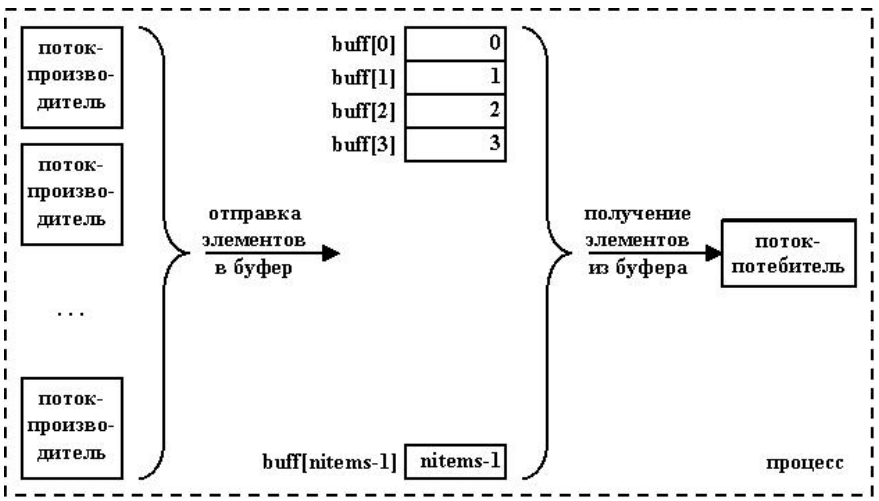

Рис. 3.16. Схема задачи производителей–потребителей

В этом первом примере важна синхронизация между потоками-производителями. Поток-потребитель не будет запущен, пока все производители не завершат свою работу. В программе prodcons_on.c, доступной на сайте в папке «Mutex», приведен текст примера.

Буфер **buff** и переменные **nput**, **nval** совместно используются потоками. Они объединены в структуру **shared** вместе с взаимным исключением, чтобы подчеркнуть, что доступ к ним можно получить только вместе с ним. Переменная **nput** хранит индекс следующего элемента массива **buff**, подлежащего обработке, а **nval** содержит следующее значение, которое должно быть в него помещено. Под структуру выделяется память и инициализируется взаимное исключение, используемое для синхронизации потоков-производителей.

Первый аргумент командной строки указывает количество элементов, которые будут произведены производителями, а второй – количество запускаемых потоков-производителей.

Каждый из создаваемых потоков-производителей вызывает функцию **produce**. Идентификаторы потоков хранятся в массиве **tid_produce**. Аргументом каждого потока-производителя является указатель на элемент массива **count**. Счетчики инициализируются значением 0, и каждый поток увеличивает значение своего счетчика на единицу при помещении очередного элемента в буфер. Содержимое массива счетчиков затем выводится на экран, так что можно узнать, сколько элементов было помещено в буфер каждым из потоков.

Программа ожидает завершения работы всех потоков-производителей, выводя содержимое счетчика для каждого потока, а затем запускает единственный процесс-потребитель. Таким образом (на данный момент) исключается необходимость синхронизации между потребителем и производителями. По завершении работы потребителя завершается работа процесса.

Критическая область кода производителя состоит из проверки на достижение конца массива (завершение работы):

```
if (shared.nput >= nitems)
```

и строк, помещающих очередное значение в массив:

```
shared.buff[shared.nput] = shared.nval;
shared.nput++; shared.nval++;
```

Эта область защищается с помощью взаимного исключения, которое разблокируется после завершения работы.

Увеличение элемента **count** (через указатель **arg**) не относится к критической области, поскольку у каждого потока счетчик свой (мас-

сив **count** в функции **main**). Эта строка не включается в блокируемую взаимным исключением область. Один из принципов хорошего стиля программирования заключается в минимизации объема кода, защищаемого взаимным исключением.

Потребитель проверяет правильность значений всех элементов массива и выводит сообщение в случае обнаружения ошибки. Эта функция запускается в единственном экземпляре и только после того как все потоки-производители завершат свою работу, так что надобность в синхронизации отсутствует.

Если убрать из этого примера (см. prodcons_off.c) блокировку с помощью взаимного исключения, он перестанет работать, как предполагается. Потребитель обнаружит ряд элементов **buff[i]**, значения которых будут отличны от i. Другой вариант его работы: несколько производителей будут формировать одни и те же элементы данных.

Также можно убедиться, что удаление блокировки ничего не изменит, если будет выполняться только один поток.

Продемонстрируем, что взаимные исключения предназначены для блокирования, но не для ожидания. Изменим (см. prodcons1.c) пример так, чтобы потребитель запускался сразу после запуска всех производителей, что позволит ему обрабатывать данные сразу по мере их формирования. Теперь придется синхронизовать потребителя с производителями, чтобы первый обрабатывал только данные, уже сформированные последними.

Начало кода (до объявления функции **main**) не претерпело никаких изменений. Поток-потребитель создается сразу же после создания потоков-производителей. Функция **produce** не изменяется. Функция **consume** вызывает новую функцию **consume_wait**. Единственное изменение в функции **consume** заключается в добавлении вызова **consume_wait** перед обработкой следующего элемента массива.

Функция **consume_wait** должна ждать, пока производители не создадут i-й элемент. Для проверки этого условия производится блокировка взаимного исключения и значение i сравнивается с индексом производителя **nput**. Блокировка необходима, поскольку **nput** может быть изменен одним из производителей в момент его проверки.

Возникает вопрос: что делать, если нужный элемент еще не готов? Здесь мы повторяем операции в цикле, устанавливая и снимая блокировку и проверяя значение индекса. Это называется опросом (spinning или polling) и является лишней тратой времени процессора.

Можно было бы приостановить выполнение процесса на некоторое время, но не известно, на какое. Что действительно нужно – это использовать какое-то другое средство синхронизации, позволяющее потоку или процессу приостанавливать работу, пока не произойдет какое-либо событие.

Взаимное исключение используется для блокирования, а условная переменная – для ожидания. Это два различных средства синхронизации, и оба они нужны. Условная переменная представляет собой переменную типа **pthread_cond_t**. Для работы с такими переменными предназначены две функции:

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cptr, pthread_
mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
```

Обе функции возвращают 0 в случае успешного завершения, положительное значение **Exxx** – в случае ошибки. Слово signal в имени второй функции не имеет никакого отношения к сигналам Unix **SIGxxx**. Просто определяется условие, уведомления о выполнении которого поток будет ожидать.

Взаимное исключение всегда связывается с условной переменной. При вызове **pthread_cond_wait** для ожидания выполнения какого-либо условия указывается адрес условной переменной и адрес связанного с ней взаимного исключения.

В программе prodcons2.c две переменные **nput** и **nval** ассоциируются с **mutex**, и мы объединяем их в структуру с именем **put**. Эта структура используется производителями.

Другая структура, **nready**, содержит счетчик, условную переменную и взаимное исключение, инициализируемое с помощью **PTHREAD_COND_INITIALIZER**. Функция **main** по сравнению с предыдущим листингом не изменяется.

Функции **produce** и **consume** претерпевают некоторые изменения. Для блокирования критической области в потоке-производителе теперь используется исключение **put.mutex**. Там же увеличивается счетчик **nready.nready**, в котором хранится количество элементов, готовых для обработки потребителем. Перед его увеличением проверяется, было ли значение счетчика нулевым, и если нет, то вызывается функция **pthread_cond_signal**, позволяющая возобновить выполнение всех потоков (здесь – потребителя), ожидающих установки ненулевого значения этой переменной.

Счетчик используется совместно потребителем и производителями, поэтому доступ к нему осуществляется с блокировкой соответствующего взаимного исключения (**nready.mutex**).

Потребитель просто ждет, пока значение счетчика **nready.nready** не станет отличным от нуля. Поскольку этот счетчик используется совместно с производителями, его значение можно проверять только при блокировке соответствующего взаимного исключения. Если при проверке значение оказывается нулевым, вызывается **pthread_cond_wait** для приостановки процесса.

При этом выполняются два атомарных действия.

1. Разблокируется **nready.mutex**.
2. Выполнение потока приостанавливается, пока какой-нибудь другой поток не вызовет **pthread_cond_signal**.

Перед возвращением управления потоку функция **pthread_cond_wait** блокирует **nready.mutex**. Если после возвращения из функции обнаруживается, что счетчик имеет ненулевое значение, то этот счетчик уменьшается (зная, что взаимное исключение заблокировано) и разблокируется взаимное исключение. После возвращения из **pthread_cond_wait** всегда заново проверяется условие, поскольку может произойти ложное пробуждение.

В примере кода функция **pthread_cond_signal** вызывалась потоком, блокировавшим взаимное исключение, относящееся к условной переменной, для которой отправлялся сигнал. В худшем варианте система немедленно передаст управление потоку, которому направляется сигнал, и он начнет выполняться и немедленно остановится, поскольку не сможет заблокировать взаимное исключение.

Исправленный код, помогающий этого избежать, будет иметь вид:

```
int dosignal;
pthread_mutex_lock(nready.mutex);
doslgnal= (nready.nready == 0);
nready.nready++;
pthread_mutex_unlock(&nready.mutex);
if (doslgnal)
pthread_cond_signal(&nready.cond);
```

Здесь сигнал условной переменной отправляется только после разблокирования взаимного исключения. Это разрешено стандартом Posix: поток, вызывающий **pthread_cond_signal**, не обязательно должен в этот момент блокировать связанное с переменной взаимное исключение. Однако Posix говорит, что если требуется предсказуемое

поведение при одновременном выполнении потоков, это взаимное исключение должно быть заблокировано процессом, вызывающим **pthread_cond_signal**.

В обычной ситуации **pthread_cond_signal** запускает выполнение одного потока, ожидающего сигнал по соответствующей условной переменной. В некоторых случаях поток знает, что требуется пробудить несколько других процессов. Тогда можно воспользоваться функцией **pthread_cond_broadcast** для пробуждения всех процессов, заблокированных в ожидании сигнала данной условной переменной.

```
#include <pttread.h>
int pthread_cond_broadcast (pthread_cond_t *aptr);
int  pthread_cond_timewait  (pthread_cond_t,  *cptr,
pthread_mutex_t *mpfr, const struct timespec *abstime);
```

Функции возвращают 0 в случае успешного завершения или положительный код **Exxx** в случае ошибки.

Функция **pthread_cond_timedwait** позволяет установить ограничение на время блокирования процесса. Аргумент **abstime** представляет собой структуру **timespec**:

```
struct timespec {
ttme_t tv_sec; /* секунды */
long tv_nsec; /*наносекунды. Заявлен на перспективу */
};
```

Эта структура задает конкретный момент системного времени, в который происходит возврат из функции, даже если сигнал по условной переменной еще не будет получен. В этом случае возвращается ошибка с кодом **ETIMEDOUT**. Этот момент представляет собой абсолютное значение времени, а не промежуток. Аргумент **abstime** задает количество секунд (наносекунды реально не поддерживаются) с 1 января 1970 UTC до того момента времени, в который должен произойти возврат из функции.

Это отличает функцию от **select, pselect** и **poll**, которые в качестве аргумента принимают некоторое количество долей секунды, спустя которое должен произойти возврат. (Функция **select** принимает количество секунд и микросекунд, **pselect** – секунд, а **poll** – миллисекунд.) Преимущество использования абсолютного времени заключается в том, что если функция возвратится до ожидаемого момента (например, при перехвате сигнала), ее можно будет вызвать еще раз, не изменяя содержимого структуры **timespec**.

При хранении взаимных исключений и условных переменных как глобальных данных всего процесса инициализировались они с помощью двух констант: **PTHREAD_MUTEX_INITIALIZER** и **PTHREAD_COND_INTIALIZER**. Инициализируемые так исключения и условные переменные приобретали значения атрибутов по умолчанию, но можно инициализировать их и с другими значениями атрибутов.

Инициализировать и удалять взаимное исключение и условную переменную можно с помощью функций

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mptr, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mptr);
int pthread_cond_init(pthread_cond_t *cptr, const pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cptr);
```

Все четыре функции возвращает 0 в случае успешного завершения работы или положительное значение **Exxx** в случае ошибки.

Аргумент **mptr** должен указывать на переменную типа **pthread_mutex_t**, для которой должна быть уже выделена память, и тогда функция **pthread_mutex_init** инициализирует это взаимное исключение. Значение типа **pthread_mutexattr_t**, на которое указывает второй аргумент функции **pthread_mutex_init (attr)**, задает атрибуты этого исключения. Если этот аргумент содержит нулевой указатель, используются значения атрибутов по умолчанию.

Атрибуты взаимного исключения имеют тип **pthread_mutexattr_t**, а условной переменной – **pthread_condattr_t**, и инициализируются и уничтожаются с помощью следующих функций:

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);
```

Все четыре функции возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

После инициализации объекта атрибутов для включения или выключения отдельных атрибутов используются отдельные функции. Один

из атрибутов позволяет использовать взаимное исключение или условную переменную нескольким процессам. Его значение можно узнать и изменить с помощью следующих функций:

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *valptr);
int pthread_mutexattr_setpshared (pthread_muutexattr_t *attr, int value);
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *valptr);
int pthread_condattr_setpshared (pthread_condattr_t *attr, int value);
```

Все четыре функции возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

Две функции **get*** возвращают текущее значение атрибута через целое, на которое указывает **valptr**, а две функции **set*** устанавливают значение атрибута равным значению **value**. Значение **value** может быть либо **PTHREAD_PROCESS_PRIVATE**, либо **PTHREAD_PROCESS_SHARED**. Последнее также называется атрибутом совместного использования процессами.

Вот как нужно инициализировать взаимное исключение для совместного использования нескольким процессам:

```
pthread_mutex_t *mptr;
pthread_mutexattr_t mattr;
mptr = malloc(sizeof (pthread_mutex_t));/* адрес */
pthread_mutexattr_init(&mattr);
pthread_mutexattr_setpshared (&mattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mptr, &mattr);
```

Здесь объявляется переменная **mattr** типа **pthread_mutexattr_t**, инициализируется значениями атрибутов по умолчанию, а затем устанавливает атрибут **PTRREAD_PROCESS_SHARED**, позволяющий совместно использовать взаимное исключение нескольким процессам. Затем **pthread_mutex_init** инициализирует само исключение с соответствующими атрибутами.

Такая же последовательность команд (с заменой **mutex** на **cond**) позволяет установить атрибут **PTHREAD_PROCESS_SHARED** для условной переменной, хранящейся в разделяемой процессами памяти.

Когда взаимное исключение используется совместно несколькими процессами, всегда существует возможность, что процесс будет

завершен (возможно, принудительно) во время работы *с* заблокированным им ресурсом. Не существует способа заставить систему автоматически снимать блокировку с ресурса во время завершения процесса. Единственный тип блокировок, автоматически снимаемых системой при завершении процесса, – блокировки записей **`fcntl`**.

Поток также может быть завершен в момент работы с заблокированным ресурсом, если его выполнение отменит (**`pthread_cancel`**) другой поток или он сам вызовет **`pthread_exit`**. Последнее сомнительно, поскольку поток должен сам знать, блокирует ли он взаимное исключение в данный момент или нет, и в зависимости от этого вызывать **`pthread_exit`**. На случай отмены другим потоком можно предусмотреть обработчик сигнала, вызываемый при отмене потока. Если же для потока возникают фатальные условия, это обычно приводит к завершению работы всего процесса.

Даже если бы система автоматически разблокировала ресурсы после завершения процесса, это не всегда решало бы проблему. Блокировка защищала критическую область, в которой, возможно, изменялись какие-то данные. Если процесс был завершен посреди этой области, что стало с данными? Велика вероятность того, что возникнут несоответствия. Если бы ядро просто разблокировало взаимное исключение при завершении процесса, следующий процесс, обратившийся к списку, обнаружил бы, что тот поврежден.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Гордеев А.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2002. – 736 с.

2. *Эпплман Д.* Windows API и Visual Basic. – М.: Русская редакция, 1999. – 926 с.

3. *Материалы* по курсу «Системное программное обеспечение» (Гунько А.В.) [Электронный ресурс] // Гунько А.В. – Режим доступа: http://gun.cs.nstu.ru/ssw

4. *Харт Дж. М.* Системное программирование в среде Windows. М.: Вильямс, 2005. – 592 с.

5. *Стивенс У.* UNIX: взаимодействие процессов. – СПб.: Питер, 2002. – 624 с.

6. *Гунько А.В.* Системное программное обеспечение. Метод. указания к лаб. работам № 3556. Новосибирск: Изд-во НГТУ, 2008. – 36 с.

ОГЛАВЛЕНИЕ

Гунько Андрей Васильевич

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Конспект лекций

В авторской редакции

Выпускающий редактор *И.П. Бровановa*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *В.Н. Зенина*

Подписано в печать 10.05.2011. Формат 60×84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 8,13. Печ. л. 8,75. Изд. № 105. Заказ № Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20